Programming Language Translator
Spring of 2003

F2
A Macromedia Flash Specification Coding Language

Authors
Benjamin Chan
Jonathan So
Shawn Tay
Jen Yu Wang

CHAPTER 1

Introduction to F2

1.1 Background

The development of information technology has revolutionized this 21st century. The existence of the World Wide Web continuously changes the life style of modern man as well as the business practice of the market. The invention of the Shockwave Flash software lies in the center of this internet age. The concept of the Internet gives rise to the idea of transporting graphics and animation across the web. Presently, over 436 million people around the globe have access to a variety of Macromedia Flash content. Macromedia Flash is a graphics, animation generation technology that provides a consistent display of graphical experience across different system platforms. Flash allows users to create scalable, interactive animation and 2 dimension graphics for the web, such as animated logos, long-form animations, navigation controls, and even complete websites, all delivered via Shockwave files (.swf, the exported Flash file format). Unlike traditional animation and movie file formats, the content and concept behind Flash is intuitive and easy to grasp for users. At the moment, Macromedia Flash and Director Programs are the two main sources available for authoring web-viewable Flash files. Both softwares rely heavily on an elaborate and expensive graphical interface. (Flash sells for $499, and Director $1,199.) Furthermore, these softwares lack portability across different operating systems (Only available for Windows and Macintosh users).

The programming language F2 is an alternative for users to produce viewable Shockwave files. This language has overcome several deficiencies that plague Macromedia Flash and Director programs. The language is designed to enable the user to program simple Flash functionalities in a non-graphical, traditional text-based programming environment. Oh yes, it's free.

1.2 Design Goal of F2

F2 is a flash graphics-purpose, object-oriented programming language. It is designed to be simple enough that it can achieve fluency similar to many commonly used flash functionalities. The F2 programming language is related to Java but is organized for the purpose of an intuitive coding approach towards flash graphics.

1.2.1 Precision

Since both Macromedia Flash and Director are graphical environments, certain operations such as the aligning, spacing, sizing and positioning of objects rely entirely on visual estimation. F2 eliminates the need to approximate during situations such as centering, horizontal line up, or exact positioning given a fix pixel number in height. F2 allows one to explicitly specify the object properties and position coordinates.

1.2.3 Availability

The availability of Macromedia Flash and Director has been greatly reduced by the expensive cost of these software. Although not as powerful and comprehensive, F2 is an open source program. It is free for all users who want to make flash graphics. Moreover, users running Linux or other popular Unix-based operating systems can overcome the inaccessibility of Macromedia programs through F2. F2 is compatible with all popular operating systems, not limited Windows and Macintosh. Built upon Java, which is extremely portable language, F2 has also inherited this nice feature of Java.

1.2.4 Simplicity

F2 is a clear, simple, and intuitive language that allows the user to construct graphics and animation through coding. Based on the concept of Macromedia Flash, F2 attempts to recapitulate the functionalities provide by Macromedia Flash without the incorporation of a user interface. In addition, it is not a graphical extension of a non-graphical language such as Java's complex graphics libraries Swing or AWT. F2 is in itself, a programming language that can produce simple Flash graphics and animation.

1.2.5 Sharable and Editable

Today, JavaScript source code is widely shared over the Internet. In fact, the complete structure behind internet websites is devoted to sharing source codes such as JavaScript. This trend supports the notion that F2 could potential become a widely-shared language like JavaScript. There are several reasons:

- As mentioned earlier, F2 is a free language. Combined with the fact that F2 is compatible with all popular operating systems, F2 can potentially capture a large number of beginner-level Flash developers.
- As a programming language, F2 incorporates into .swf movies the ease in editing that is only natural to text-based source codes. The difficulty in editing .fla (the editable Flash file format) files through a complex graphical interface gives F2 a comparative advantage.

CHAPTER 2 F2 Tutorial

F^2 is a simple but powerful animation language. In general, the language and syntax is very similar to Java, especially for variable declarations, arrays, for loops, while loops, and even object and frame declarations. Semi-colons are used as separators.

To begin writing F^2, it is important to know the grammatical structure of each animation or movie. Each movie must start with the Header Statements, which are shown in Figure 1, lines 3 to 8. These specify the general characteristics of the movie. The sample program below outputs a movie called *mymovie*.swf, which has a width and height of 100 pixels, plays at 12 frames per second, contains 100 frames total, and has a white background color (refer to the LRM for the complete list of colors). Also, the Header variables must be declared in capital letters.

```
1.  import "othermovie";
2.  import "othermovie2";
3.  MOVIENAME = "mymovie";
4.  WIDTH = 100;
5.  HEIGHT = 100;
6.  FPS = 12;
7.  NUMFRAMES = 100;
8.  BGCOLOR = WHITE;
9.
10. Movie {
11.
12.
13.     Frame (50) {
14.
15.
16.     }
17.    Frame (50) {
18.
19.    }
20.    }
```

Fig. 1 - Program Structure

After the Header we are ready to start declaring variables, creating objects, and adding them to the frames of our movie. All this happens in the Movie body, as declared on line 10 (the end brace is on line 17), in Figure 1. It is important to note that all variable and object declaration must be declared *before* any Frame blocks, for example within lines 11 and 12 in the sample program. Some declarations are shown in Figure 2 below.

```
1. int x = 12;
2. String y = "jon";
3. Text myText = new Text (y, 30, BLUE);
4. Circle myCirc = new Circle(10, RED);
```

Fig. 2 – Variable and Object Declarations

Now that we have created a circle (line 4 of Figure 2) and some text (line 3), we are ready to place them into frames. All of this happens within the Frame block declaration, as shown in Figure 1, line 13. The integer 12 specifies the number of frames defined within the block. The first frame block defines the first 50 frames in the 100 frame total movie, while the next frame block (line 17) defines the next 50 frames. Within a Frame block three actions can be performed: Place, Animate, and Insert. Place is used to put objects on the screen, Animate is used to linearly move objects, and Insert is used to insert imported movies. Some frame block definitions are shown below in Figure 3.

```
1. Frame (50) {
2.     Place (myCirc, 0, 0, 1, 25, 1);
3.     Place (myText, 0, 0, 26, 50, 2);
4. }
5.
6. Frame (50) {
7.     Animate(myCirc, 0, 0, 1, 85, 85, 20, 1);
8.    Insert ("othermovie", 0, 0, 21, 2);
9. }
```

Fig. 3.- Frame block declarations and definitions

The sample code in Figure 3 shows our *myCirc* (line 2) object on the screen for frames 1-25 of the movie, and then shows our *myText* (line 3) object for the next 25 frames. Line 7 moves our *myCirc* object from the top left corner to the bottom right corner of the movie for the next 20 frames.

The last feature we need to discuss is importing new movies. Suppose we have a previously created movie (source code, the .f2 file), we can import the movie into our current movie. This is shown in Figure 1, line 1 and 2, with a file called *othermovie*.f2 and *othermovie2*.f2. Import statements must be made before the header, and any number of them can be made. Our *othermovie* has been inserted into the last 30 frames of our movie in Figure 3, line 8.

Finally, to generate a shockwave movie, save your program with the extension *.f2*, and then at the command line run *java F2_1Main moviename.f2*. If there are no errors, you should receive a confirmation of a successful compilation. This concludes the tutorial. Happy programming!

CHAPTER 3

Reference Manual

3.1 Introduction

F2 is a computer language built with the intention to integrate programming capability into the generation of shockwave flash files. The ultimate goal for F2 is to overcome Macromedia's lack of precision, reproducibility, and portability of SWF files. Contrary to Macromedia Flash generator, F2 is a very powerful tool for users who desire a faster approach to flash generation than the conventional macromedia software. The syntax behind F2 is designed to capture an appropriate translation from graphical design to written code. The complete grammar of F2 integrates Java's strictly typed structure approach to extrapolate further a language base appropriate for flash generation.

3.2 Grammar

This section describes the context-free grammars used in F2 specification to define the lexical and syntactical structure of a program.

3.2.1 Grammar Notation

Lexical and syntactic grammars for F2 are introduced in this specification. The lexical grammar has as its terminal symbol the characters of the Unicode character set. It defines a set of productions, starting from distinguished nonterminals that describe how sequences of Unicode characters are translated into a sequence of input elements. These input elements, with white space and comments discarded, form tokens. The syntactic grammar describes how sequences of tokens can form syntactically correct programs.

Non-terminal and terminal symbols are shown in separate fonts in the production of the lexical and syntactic grammars, and throughout this specification whenever the text is directly referring to such symbols. *Non-terminal* symbols are shown in *italic* type. The definition of a *Non-terminal* is introduced by the name of the *Non-terminal* being defined followed by a colon. On the other hand, terminal is shown in fixed width font.

Symbols are quoted to denote terminals, separating from symbols used for the purposes of good formatting throughout the text. S? denotes that the

symbol S is optional. S* denotes that the symbol S may occur zero or more times. S+ denotes that the symbol S will occur one or more times. (S|T) denotes a choice between the symbol sequences S and T.

3.3 Lexical Convention

An F2 program can be stored as a stand alone movie file or it can *import* multiple F2 movie files to become a composite movie file. A copy of each of these *import* units is stored into the program upon the calling of the embedded Insert function. Programs are written using the Unicode character set accepted by the Java Virtual Machine which will be used to run the F2 compiler.

3.3.1 Line Terminators

Lines are terminated by the ASCII characters CR, or LF, or CR LF. The two characters CR immediately followed by LF are counted as one line terminator, not two.

LineTerminator:
The ASCII LF, also known as "newline"
The ASCII CR, also known as "return"
The ASCII CR followed by the ASCII LF

3.3.2 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators and other separators. Blanks, horizontal tabs, newlines, form feeds, and comments as described later are ignored, except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants. If the input stream has been separated into tokens up to a given characters, the next tokens is the longest string of characters that could constitute a token.

3.3.3 Comments

There are two kinds of comments:

/* text */	A traditional comment: all the text from the ASCII characters /* to the ASCII characters */ is ignored.
// test	A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored.

Comments do not nest and they do not occur within string or character literals.

3.3.4 White Space

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.

3.3.5 Identifiers

An *identifier* is an unlimited-length sequence of F2 letters and F2 digits, the first of which must be a letter. An identifier cannot have the same spelling as a keyword, Boolean literal, or the null literal. F2 interprets underscore (_) as a letter and it is case sensitive.

3.3.6 Literals

Integer Literals

An F2 integer literal is a decimal numeral. A decimal numeral is either the single ASCII character Ø, representing the integer 0, or consists of an ASCII digit from 1 to 9, optimally followed by one or more ASCII digits from Ø to 9, representing positive integers.

String Literals

A string literal, also called a string constant, is a sequence of characters surrounded by double quotes, as in "…". Adjacent string literals are concatenated into a single string. String literals do not contain newlines or double-quote characters; in order to represent them, escape sequences are available.

Newline	\n	Form feed	\f
Carriage return	\r	question mark (?)	\?
Horizontal tab	\t	double quote (")	\"
Backslash (\ )	\\		

Boolean Literals

The Boolean type has two values, represented by the literals true and false, from ASCII letters.

Null Literals

The null type has one value, the null preference, represented by the literal null, which is formed from ASCII characters. A *null literal* is always the null type.

3.3.7 Constant

A constant consists of a sequence of decimal digits. A constant is unsigned and never negative. The maximum value for a constant is 2^{32}.

3.3.8 Keywords

There are three sets of keywords: First set has keywords with initial uppercase letter. Second set has keywords with all lowercase letters. Third set has keywords with all uppercase letters. The following identifiers are reserved for use as the three sets of keywords, and cannot be used otherwise:

FPS	Animate	break
HEIGHT	Circle	do
MOVIENAME	Ellipse	else
NUMFRAMES	Frame	for
WIDTH	Import	if
BGCOLOR	Insert	int
BLACK	Line	return
BLUE	Movie	while
BROWN	Object	
GREEN	Place	
GREY	Random	
ORANGE	Rectangle	
PURPLE	String	
RED		
WHITE		
YELLOW		

While true and false might appear to be keywords, they are technically Boolean literals. Similarly, while null might appear to be a keyword, it is technically the null literal.

3.4 Meaning of Identifiers

Identifiers, or names, refer to a variety of things: functions and objects. An object, sometimes called a variable, is a location in storage and its interpretation depends on its type.

3.4.1 Basic Data Types

There are two fundamental types. Objects, declared as integers (int), contain unsigned decimal digits with a maximum value of up to 2^{32}. Objects, declared as strings (String), contain a sequence of characters.

3.4.2 Boolean Type

The Boolean type represents a logical quantity with two possible values, indicated by the literals true and false.

3.4.3 Reference Types

Besides the basic data types, there are two kinds of reference types:

Arrays of objects of a given type;
Methods returning objects of a given type;

In general these methods of constructing objects can be applied recursively.

3.4.4 Objects

An Object can be of type atomic shape object or composite shape object. These two types of shape objects are defined as follows:

Atomic Shape Objects

There are five atomic shape objects:

Text
Circle
Ellipse
Line
Rectangle

For detailed descriptions, refer to sections dedicated to the above atomic shape objects.

Composite Shape Object

There is one composite shape object:

Object

A composite shape object, defined by the user, contains a series of atomic shape objects and other composite shape objects previously defined by the user.

3.4.5 Variables

A variable is a storage location and has an associated type. A variable always contains a value that is assignment compatible with its type. Moreover, every variable in a program must have a value before its value is used.

3.5 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, the highest precedence first. Within each subsection, the operators have the same precedence. Left- or right-associative is specified in each subsection for the operators discussed therein.

3.5.1 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

Primary-expression: *Identifier-primary*
| *constant*
| *"true"*
| *"false"*
| *"null"*
| *New-expression*
| (*Assignment-expression)*

An identifier is a primary expression, provided that it has been suitably declared as discussed below. Its type is specified by its declaration.
A constant is a primary expression, with either type int or type string.
New-expression is an initialization expression that declares the instance of an array object.
A parenthesized assignment expression is a primary expression that contains operators.

3.5.2 Multiplicative Operators

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

Multiplication-expression: *Multiplication-expression * Postfix-expression*
| *Multiplication-expression / Postfix-expression*
| *Multiplication-expression % Postfix-expression*

The operands of * and / must have arithmetic type; 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 yields the quotient and the % operator the remainder, of the division of the first operand by the second; if the second operand is 0, the result is undefined.

3.5.3 Additive Operators

The additive operators + and – group left-to-right. If the operands have arithmetic type, the usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

Additive-expression: *Multiplicative-expression*
| *Additive-expression* + *Multiplication-expression*
| *Additive-expression* – *Multiplication-expression*

String Concatenation Operator +

If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a newly created String that is the concatenation of the two operands strings. The left-hand String operand precedes the right-hand operand in the newly created String.

Additive Operator (+ or -) for Integers

The result of the + operator is the sum of the operands. The result of the – operator is the difference of the operands.

3.5.4 Relational Operators

The relational operators group left-to-right.

Relational-expression: *Relational-expression* < *Additive-expression*
| *Relational-expression* > *Additive-expression*
| *Relational-expression* <= *Additive-expression*
| *Relational-expression* >= *Additive-expression*

The type of each of the operands must be of an integer type or a compile time error occurs. The operators < (less), > (greater), <= (less or equal) and >= (greater or equal) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int.

3.5.5 Equality Operators

The equality operators are syntactically left-associative.

Equality-expression: *Relational-expression*
| *Equality-expression* == *Relational-expression*
| *Equality-expression* != *Relational-expression*

If the operands are of integer type, a numeric equality test is performed. If the operands are both type Boolean, then the operation is Boolean equality. The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.

3.5.6 Assignment Expressions

Assignment-expression: *Primary-expression*
| *Identifier* = *Assignment-expression*

The result for the first operand must be a variable, or a compile time error occurs. This operand must be a named variable. The type of the assignment expression is the type of the variable. In the assignment, with =, the value of the expression replaces that of the object referred to by the *Identifier*.

3.6 Declarations

Declaration: *TypeSpecifier InitIdentifierList* ;

TypeSpecifier: int
| String

InitIdentifierList: *InitIdentifier*
| *InitIdentifierList* , *InitIdentifier*

InitIdentifer: *Identifier*
| *Identifier* = *StringExpression* ;
| *Identifier* = *IntExpression* ;

A declaration consists of a *TypeSpecifier*, followed by an *identifier*, and possibly followed by an equal sign and a *StringExpression* or an *IntExpression* (if the user chooses to declare and initialize.

3.6.1 Array Declarations for Integers and Strings

ArrayDeclaration: *TypeSpecifier ArrayIdentifier* ;

ArrayIdentifier: *identifier BracketList*
| *identifier EmptyBracketList InitArrayIdentifier*

BracketList: [*IntExpression*]
| *BracketList* [*IntExpression*]

EmptyBracketList: [ ]
| *EmptyBracketList* [*IntExpression*]

InitArrayIdentifier: = { *IntExpressionList* }
| = { *StringExpressionList* }

IntExpressionList: *IntExpression*
| *IntExpressionList* , *IntExpression*

StringExpressionList: *StringExpression*
| *StringExpressionList* , *StringExpression*

3.6.2 Array Declarations for AtomicObjects

AtomicObjectArrayDeclaration: *AtomicObjectType AtomicObjectArrayIdentifier* ;

AtomicObjectArrayIdentifier: *Identifier* [*IntExpression*]

When declaring an array of *AtomicObjects,* each atomic object in the array is automatically initialized with default values specific to its atomic object type.

3.6.3 Function Declarations and Definitions

Functions are declared and defined outside of the Movie block. They are type-specified by an *AtomicObjectType*, and int, or a String. Following the *TypeSpecifier* is the *identifer*, and following the *identifier* is the parameter list surrounded by parentheses.

ParameterTypeList: *ParameterList*
| *ParameterList* , . . .

ParameterList: *ParameterDeclaration*
| *ParameterList* , *ParameterDeclaration*

ParameterDeclaration: *AtomicObjectType identifier*
| String *identifier*
| int *identifier*

AtomicObjectType: Text
| Rectangle
| Circle
| Ellipse
| Line

The function is then defined within a pair of braces. At the end of the function definition, the function returns an atomic object, string, or integer.

3.7 Statements

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

Statement: *Selection-statement*
| *Iteration-statement*
| *Jump-statement*

3.7.1 Selection Statements

Selection statements choose one of several flows of control.

Selection Statements: if (*expression*) *Statement*
| if (*expression*) *Statement* else *Statement*

In both forms of the if statements, the expression is evaluated, and if it compares unequal to 0, the first sub statement is executed. In the second form, the second sub statement is executed if the expression is 0. The else ambiguity is resolved by connecting an else with the last encountered else-less if at the same block nesting level.

3.7.2 Iteration Statements

Iteration statements specify looping.

Iteration-statement: while (*expression*) Statement
| for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$)
Statement

In the while statement, the sub statement is executed repeatedly so long as the value of the expression remains unequal to 0; the expression must be

arithmetic. With while, the test occurs before each execution of the statement.
In the for statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on it type. The second expression must be arithmetic; it is evaluated before each iteration, and if it becomes equal to 0, the for is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type.

3.8 F^2 Source File Specifications

This section describes the format of an F^2 source file. More specifically, it describes the import statement, header statements, Movie block, atomic object declarations, composite object declarations, Frame block, place declaration, animate declaration, and insert declaration.

3.8.1 Import Statement

ImportDeclaration: import *StringExpression* ;

The import statement tells the compiler to prepare a data structure based upon another program (filename is the *StringExpression*) for use by the current program. This is done before any further processing of the current program. All import statements must be placed at the very beginning of the program before any other commands.

The import feature allows the user to reuse movies that they have already created. However, the imported movie must be used in its entirety. When a movie is reused, the movie will be placed relative to the coordinates specified by the user. Also the imported movie inherits current movie's background color, and frames per second. Checks will ensure that the imported movies are no larger in length and width than the current movie, and that the number of frames in the imported movie does not exceed the number of frames in the Frame block of the current movie.

3.8.2 Header Statements

MovieDeclaration: MOVIENAME = *StringLiteral* ;

FrameWidthDeclaration: WIDTH = *IntExpression* ;

FrameHeightDeclaration: HEIGHT = *IntExpression* ;

FPSDeclaration: FPS = *IntExpression* ;

NumFramesDeclaration: NUMFRAMES = *IntExpression* ;

BGColorDeclaration: BGCOLOR = *ColorExpression* ;

ColorExpression:
BLACK
| BLUE
| BROWN
| GREEN
| GREY
| ORANGE
| PURPLE
| RED
| WHITE
| YELLOW

The header statements are where the user defines the general setting for the movie. They have to be declared in the shown order, none can be omitted or set to null values, and they all must be declared before the Movie block.

MOVIENAME defines the name that the .swf file will be saved as. This name does not have to match the name of the source-code file. The name must be a sequence of letters and digits. The first character must be a letter.
FPS defines the rate at which the movie will be played in frames per second, and must be a non-negative integer.
NUMFRAMES defines the number of frames for the entire movie, and must be a non-negative integer.
WIDTH defines the width of the movie in pixels, and must be a non-negative integer.
HEIGHT defines the height of the movie in pixels, and must be a non-negative integer.
BGCOLOR defines the background color of the movie, and will stay constant for the entire length of the movie. The color specified must be one of the ten recognized standard colors: RED, BLUE, GREEN, WHITE, ORANGE, YELLOW, BLACK, PURPLE, BROWN and GRAY.

3.8.3 Movie Block Definition

Movie: Movie { *CompoundStatement* }

CompoundStatement: *DeclarationList StatementList*

StatementList: *Statement*
| *StatementList Statement*

DeclarationList: *Declaration*
| *DeclarationList Declaration*

Declaration: *AtomicObjectDeclaration*
| Composite*ObjectDeclaration*
| *PrimitiveTypeDeclaration*

PrimitiveTypeDeclaration: String *IdentifierList* ;
| int *IdentifierList* ;

IdentifierList: *Identifier*
| *IdentifierList* , *Identifier*

The Movie block definition is similar to the "main" method declaration in a Java program. Object and frame declarations will be made within the braces. The import and header statements must be made outside and before the Movie block definition, and nothing may follow the definition either.

3.8.4 Atomic Object Declarations

AtomicObjectDeclaration: Text *identifier* (*TextParam*) ;
| Rectangle *identifier* (*RectangleParam*) ;
| Circle *identifier* (*CircleParam*) ;
| Ellipse *identifier* (*EllipseParam*) ;
| Line *identifier* (*LineParam*) ;

TextParam: *StringExpression* , *IntExpression* , *ColorExpression*

RectangleParam: *IntExpression* , *IntExpression* , *ColorExpression*

CircleParam: *IntExpression* , *ColorExpression*

EllipseParam: *IntExpression* , *IntExpression* , *ColorExpression*

LineParam: *IntExpression* , *IntExpression* , *IntExpression* ,
IntExpression , *ColorExpression*

The primitive object declarations are used to create atomic (no other previously created objects can be appended to it) objects. The five primitive objects from which all other objects will be built are listed above, along with their respective parameters.

The Text parameters are as follows: the text, font size, color.

The Rectangle parameters are as follows: width, height, color.
The Circle parameters are as follows: radius, color.
The Ellipse parameters are as follows: width, height, color.
The Line parameters are as follows: start x, start y, end x, end y, color.

The limit for the size of the object is the size of the movie itself, which cannot be exceeded. The atomic object declarations must be declared within the Movie block definition but outside of the Frame block definitions.

3.8.5 Frame Block Definitions

FrameDefinition: Frame (*IntExpression*) { *StatementList* }

The Frame block definitions are used to dictate what will happen within the amount of frames indicated by the integer value, which can be an integer identifier or constant, but must be non-negative. Within the Frame block definitions are any number of Place, Animate, and Append declarations. It is possible to leave the body empty however. The order in which the Frame blocks are declared is the order in which they will be played:

Frame (100){<body1>}
Frame (50){<body2>}
Frame (300) {<body3>}

The above code means that from frames 0 to 99, body1 will be played. From frames 100 to 149, body2 will be played. From frames 150 to 449, body3 will be played.

Any number of frame block definitions are allowed, but the total number of frames must not exceed the number of frames in the movie. If they do, then an error will be given and the file will not compile properly.

3.8.6 Place Declaration

PlaceDeclaration: Place (*StringLiteral* , *IntExpression* , *IntExpression* , *IntExpression* , *InteExpression* , *IntExpression*) ;

The parameters are: object name, x coordinate, y coordinate, start frame, end frame, depth.

The place declaration is used to put objects in the frames at the specified location and depth for a set number of frames. The start and end frames

dictate how long and when the object will be shown. The string can be a string identifier or constant, and the integers must be non-negative. The position for placing the object must be within the size of the movie, and the start and end frame interval must not exceed the number of frames declared in the frame block definition in which the place declaration is located. A depth of 0 is at the top, and the higher the depths, the lower the object on the screen. The place declaration can only be declared within frame block bodies.

3.8.7 Animate Declaration

AnimateDeclaration: Animate (*StringLiteral* , *IntExpression* , *IntExpression* , *IntExpression* , *IntExpression* , *IntExpression* , *IntExpression*) ;

The parameters are: object name, start x coordinate, start y, start frame, end x coordinate, end y, end frame, depth.

The animate declaration is used to create linear motion. The user will say the start position, the end position, and the frame interval for motion to occur. The program will then extrapolate the line of motion and the position of the object in each frame. This functionality makes it easy to move things linearly, from one spot to another over as many frames as needed. The string can be a string identifier or constant, and the integers must be non-negative. The position for placing the object must be within the size of the movie, and the start and end frame interval must not exceed the number of frames declared in the Frame block definition in which the place declaration is located. A depth of 0 is at the top, and the higher the depths, the lower the object on the screen. The animate declaration can only be declared within Frame block bodies.

3.8.8 Insert Declaration

InsertDeclaration: Insert (*StringLiteral* , *IntExpression* , *IntExpression* , *IntExpression* , *IntExpression*) ;

The parameters are: movie name, relative x coordinate, relative y coordinate, start frame, depth.

The insert declaration is the functionality that allows the user to use the movies that have been imported using the import declaration. The relative x and y coordinates define where the imported movie should be placed in relation to the current movie. The start frame dictates when the imported movie should start playing. There will be a check that ensures the

imported movie length does not exceed the length defined by the Frame block definition. . The string can be a string identifier or constant, and the integers must be non-negative. The position for placing the object must be within the size of the movie. A depth of 0 is at the top, and the higher the depths, the lower the object on the screen. The insert declaration can only be declared within Frame block bodies.

CHAPTER 4

Project Plan/Schedule

4.1 Project Responsibilities

The project is evenly divided among the team members to achieve maximum efficiency. Nevertheless, testing and debugging of all project code are done together by all team members. The development task of F2 is divided accordingly:

Benjamin Chen	Backend, Code Generation, Testing
Jonathan So	Tree Walking, Code Generation, Testing
Shawn Tai	Lexer, Parser, Tree Walking, Testing
Jen Yu Wang	Lexer, Parser, Documentation, Code Generation

4.2 Prospective Project Development Schedule

The following deadlines were proposed for various key development phases.

2/18/03	Produce F2 language whitepaper, outlining rough language features and design goals for F2.
3/15/03	Establish implementation approach, development environment, and code platform/convention.
3/24/03	Meet with Prof. Edwards to discuss potential problems and details of the proposed approach to F2's development.
3/27/03	Produce Language Reference Manual.
4/05/03	Complete lexer, parser, and tree walker.
4/12/03	Complete code generation.
4/26/03	Complete compiler.
5/03/03	.swf files simulation.
5/09/03	Final testing and debugging
5/13/03	Code freeze, project feature complete.

4.3 Software Development Environment

This project is developed in UNIX using Java SDK 1.4.1. ANTLR version 2.7.2 is the utility used to develop the F2 parser. The F2 scanner is a production of a set of Java programs. The testing of F2 files will be done by using Macromedia Flash Player. Makefiles are created accordingly in every source directory.

4.4 Project Log

The following log has recorded the actual dates on significant development breakthroughs.

2/05/03	Project Initiated
2/18/03	Language Whitepaper completed
3/07/03	First draft of F2 code convention
3/14/03	Development Environment established and outlined.
3/14/03	Produced target flash file for project to achieve
3/20/03	Grammar, first draft
3/27/03	First draft of language reference manual
4/05/03	Testing Phase I initiated
4/12/03	Code lexical and syntactical grammar finalized
4/23/03	First working version of lexer and parser
03/31/03	First working version of compiler back-end
4/04/03	Starting Code generation
4/04/03	First working version of intermediate representation
5/03/03	Work on code checking in Macromedia Flash Player
05/10/03	Finalized version of scanner Finalized version of compiler
05/03/03	Sample .swf produced under F2
05/10/03	Simulation of Macromedia Flash using F2
05/13/03	Final Paper
5/14/03	Meeting With Prof. Edwards

CHAPTER 5

Architecture Design

The structure of the F2 compiler is based upon the foundation of Macromedia Flash software. Given that the objective behind F2 is to produce a programmable Flash language that can generate Macromedia Flash Shockwave files, it is thus absolutely necessary for F2 to work around the structural representation embedded within Macromedia.

In this project, the F2 compiler is broken down into the following parts: lexer, parser, tree walker, runtime environment, and code generator.

Front-End: Lexer, Parser, and Tree Walker

Starting with the front end, F2 has been modeled after a graphics user interface. It has been organized accordingly to form a clear vocabulary (as shown in Chapter 2 Tutorial) to abridge the gap between words and pictures. Grammar features are thoroughly explained in Chapter 3.

The concept behind an F2 movie is that it is a collection of individual frames, with each frame containing the necessary objects that hold their individual specifications. The data structure behind the implementation of this concept is that each Movie file contacts an array of frames, with each frame containing a Vector to hold its objects to be shown. And each Shape object is defined as objects with their own necessary parameters. The parsing of the F2 language is implemented around this central theme. With the defined parser rules, parsing is done within the extended tree walker. Inside of the tree walker, F2 create integer or string variables and assign values as instructed by the grammar rules. Integer evaluation is also performed locally inside of the walker before assignment occurs. Upon hitting a Shape object declaration, the walker instantiates the defined abstract object according to the type parsed by the walker. And the walker will assign the parameters for this object locally. Since F2 is a strongly type language, type checking is enforced strictly during compile time, this measure is done to prevent runtime errors derived from incompatible type assignment or evaluation.

At the end of the tree walker, depending on whether the key word “import” is called or not, the tree walker will pass filled the array of frame and frame vectors onto the next stage, Runtime Environment and Code Generation.

Runtime Environment and Code Generation: ImportBox.java, BlackBox.java, WriteSWF.java

As mentioned before, at the end of the tree walker, an array of Vector frames has been passed onto BlackBox.java. However, before BlackBox starts execution, the walker will decide first whether to import a separate F2 file or not. This is decided by the occurrence of the key word "import." If the import key word does exist in the original F2 file, the tree walker will then call ImportBox.java as the Runtime environment to compile the indicated F2 import movie. The import movie is then parsed with its frames stored in the array and objects stored in their corresponding vectors. At the end of ImportBox, it returns the array of Frame Vector and pass it to BlackBox.

The next stage is the BlackBox, where detailed algorithms for specific F2 functionalities are written. (Functions such as Place(), Insert(), Animate() have been implemented inside of BlackBox. Their algorithms involve mostly flash tag calculations.) In general, the BlackBox if necessary, just takes in an (or multiples of, depend on how many movies are imported) array of Frame Vector and makes the necessary calculations for the called functions. At the end of a series of evaluation and reassignment of values, BlackBox loops through the array of Frame Vectors from import F2 movies and incorporates it into its own generated array of Frame Vectors. It then starts assembly code generation by looping through the array. Once the assembly code generation is done, BlackBox calls WriteSWF.java to translate everything into the binary representation of F2 with the file extension .swf.

A side note to users is that F2's assembly language is an important feature. This one extra layer of abstraction between source code and binary bits grant the users with more power. Given that Macromedia has already established a system of binary tag representation for its graphics elements. F2 inherits these tags and organize these tags into an intermediate representation, which can be easily read by the users. The simplicity and structured format of the assembly code allows easy debugging for later.

Assembly Language Specification

```
/**
 * Assembly Language for F^2
 *
 * Benjamin Chan
 * 05/05/03
 */

// File Header in every flash file
SwfFileHeader
  F                 // always F
  W                 // always W
  S                 // always S
  Version 6         // 1-6
  FileLength 225    // Length of file in bytes
  RECT              // RECT data type (Background size)
    Nbits 15
    Xmin  0
    Xmax  11000
    Ymin  0
    Ymax  8000
  FrameRate 12      // fps
  NumFrames 13      // total number of frames

// End Tag
End
  Tag 0
  Length 0

// ShowFrame Tag
ShowFrame
  Tag 1
  Length 0          // Length of Tag in bytes
                    // Either "Length" or "Long Length" ( > 63 bytes)

// DefineShape Tag
DefineShape
  Tag 2
  LongLength 73     // Either "Length" or "Long Length" ( > 63 bytes)
  ShapeID 1         // ID of the Shape in the Dictionary
  RECT              // RECT data type (Shape Size)
    Nbits 12
    Xmin -1170
    Xmax 1170
    Ymin -1170
    Ymax 1170
  ShapeWithStyle
    FillStyleArray
      FillStyleCount 5 // Either "FillStyleCount" or "FillStyleCountExtended"
      FillStyleType 0  // 5 Different FillStyleTypes; 0 = solid fill
        RGB         // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
          Red 0
          Green 102
          Blue 204
      FillStyleType 16  // 16 = linear gradient fill
        Matrix          // Matrix data type
          hasScale 1
            NScaleBits 12  // if hasScale == 1
            ScaleX 1170    // if hasScale == 1
            ScaleY 1170    // if hasScale == 1
          hasRotate 1
```

```
      NRotateBits 12 // if hasRotate == 1
      RotateSkew0    // if hasRotate == 1
      RotateSkew1    // if hasRotate == 1
    NTranslateBits 15
    TranslateX 9478
    TranslateY 5877
  Gradient
    NumGradients 2 // Number of GradRecords
    GradRecords
      Ratio 5
      RGBA       // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
        Red 0
        Green 102
        Blue 204
        Alpha 100
    GradRecords
      Ratio 8
      RGBA       // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
        Red 0
        Green 201
        Blue 102
        Alpha 80
FillStyleType 18  // 18 == radial gradient fill
  Matrix             // Matrix data type
    hasScale 1
      NScaleBits 12  // if hasScale == 1
      ScaleX 1170    // if hasScale == 1
      ScaleY 1170    // if hasScale == 1
    hasRotate 1
      NRotateBits 12 // if hasRotate == 1
      RotateSkew0    // if hasRotate == 1
      RotateSkew1    // if hasRotate == 1
    NTranslateBits 15
    TranslateX 9478
    TranslateY 5877
  Gradient
    NumGradients 2 // Number of GradRecords
    GradRecords
      Ratio 5
      RGBA       // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
        Red 0
        Green 102
        Blue 204
        Alpha 100
    GradRecords
      Ratio 8
      RGBA       // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
        Red 0
        Green 201
        Blue 102
        Alpha 80
FillStyleType 64  // 64 == tilted bitmap fill
  BitmapID 2     // ID of bitmap character for fill
  Matrix             // Matrix data type
  hasScale 1
    NScaleBits 12  // if hasScale == 1
    ScaleX 1170    // if hasScale == 1
    ScaleY 1170    // if hasScale == 1
  hasRotate 1
    NRotateBits 12 // if hasRotate == 1
    RotateSkew0    // if hasRotate == 1
    RotateSkew1    // if hasRotate == 1
  NTranslateBits 15
  TranslateX 9478
```

```
    TranslateY 5877
  FillStyleType 65  // 65 == tilted bitmap fill
    BitmapID 2     // ID of bitmap character for fill
    Matrix          // Matrix data type
    hasScale 1
      NScaleBits 12  // if hasScale == 1
      ScaleX 1170    // if hasScale == 1
      ScaleY 1170    // if hasScale == 1
    hasRotate 1
      NRotateBits 12 // if hasRotate == 1
      RotateSkew0    // if hasRotate == 1
      RotateSkew1    // if hasRotate == 1
    NTranslateBits 15
    TranslateX 9478
    TranslateY 5877
LineStyleArray
  LineStyleCount 2 // Either "LineStyleCount" or "LineStyleCountExtended"
  Width 20
  RGB             // RGB (if Shape1 or Shape2) or RGBA (if Shape3)
    Red 0
    Green 0
    Blue 0
NumFillBits 1
NumLineBits 1
ShapeRecord
  StyleChangeRecord
    TypeFlag 0       // always 0 for StyleChangeRecord
    StateNewStyles 1
    StateLineStyle 1
    StateFillStyle1 1
    StateFillStyle0 1
    StateMoveTo 1
    MoveBits 11      // if StateMoveTo == 1
    MoveDeltaX 820    // if StateMoveTo == 1
    MoveDeltaY 820    // if StateMoveTo == 1
    FillStyle0 1     // if StateFillStyle0 == 1
    FillStyle1 1     // if StateFillStyle1 == 1
    LineStyle 1      // if StateLineStyle1 == 1
    FillStyleArray    // if StateNewStyles == 1
      ...
    LineStyleArray    // if StateNewStyles == 1
      ...
    NumFillBits 1     // if StateNewStyles == 1
    NumLineBits 1     // if StateNewStyles == 1
  StraightEdgeRecord
    TypeFlag 1       // always 1 for StraightEdgeRecord
    StraightFlag 1    // always 1 for StraightEdgeRecord
    NumBits 11
    GeneralLineFlag 1 // GeneralLine == 1; Vert/Horz Line == 0
    DeltaX 820       // if GeneralLineFlag == 1
    DeltaY 820       // if GeneralLineFlag == 1
    VertLineFlag     // if GeneralLineFlag == 0; if 0->Vert, 1->Horz
    DeltaX          // if GeneralLineFlag == 0, and VertLineFlag == 0
    DeltaY          // if GeneralLineFlag == 0, and VertLineFlag == 1
  CurvedEdgeRecord
    TypeFlag 1       // always 1 for CurvedEdgeRecord
    StraightFlag 0    // always 0 for CurvedEdgeRecord
    NumBits 8
    ControlDeltaX 685
    ControlDeltaY 340
    AnchorDeltaX 543
    AnchorDeltaY 0
  EndShapeRecord
    TypeFlag 0
```

```
        EndOfShape 0

// PlaceObject Tag
PlaceObject
  Tag 4
  Length 10             // Either "Length" or "Long Length" ( > 63 bytes)
  CharacterID 1
  Depth 1
  Matrix
    ...
  CXForm
    hasAddTerms 1
    hasMultTerms 1
    NBits 4
    RedMultTerms 10     // if hasAddTerms == 1
    GreenMultTerms 10   // if hasAddTerms == 1
    BlueMultTerms 10    // if hasAddTerms == 1
    RedAddTerms 10      // if hasMultTerms == 1
    GreenAddTerms 10    // if hasMultTerms == 1
    BlueAddTerms 10     // if hasMultTerms == 1

// RemoveObject Tag
RemoveObject
  Tag 5
  Length 10             // Either "Length" or "Long Length" ( > 63 bytes)
  CharacterID 1
  Depth 1

// SetBackgroundColor Tag
SetBackgroundColor
  Tag 9
  Length 10             // Either "Length" or "Long Length" ( > 63 bytes)
  RGB
    ...

// DefineShape2 Tag
DefineShape2
  Tag 22
  Length 10             // Either "Length" or "Long Length" ( > 63 bytes)
  ShapeID 1
  RECT
    ...
  ShapeWithStyle
    ...

// Protect Tag
Protect
  Tag 24
  Length 0

// PlaceObject2 Tag
PlaceObject2
  Tag 26
  Length 10             // Either "Length" or "Long Length" ( > 63 bytes)
  PlaceFlagHasClipActions 0  // always 0, coz tag not written
  PlaceFlagHasClipDepth 0    // always 0, coz tag not written
  PlaceFlagHasName 1
  PlaceFlagHasRatio 1
  PlaceFlagHasColorTransform 1
  PlaceFlagHasMatrix 1
  PlaceFlagHasCharacter 1
  PlaceFlagMove 1
  Depth 1
  CharacterID 1         // if PlaceFlagHasCharacter == 1
```

```
  Matrix              // if PlaceFlagHasMatrix == 1
    ...
  CXForm              // if PlaceFlagHasColorTransform == 1
    ...
  Ratio 5             // if PlaceFlagHasRatio == 1
  Name                // if PlaceFlagHasName == 1
    ...

// RemoveObject2 Tag
RemoveObject2
  Tag 28
  Length 10           // Either "Length" or "Long Length" ( > 63 bytes)
  Depth 1

// DefineShape3 Tag
DefineShape3
  Tag 32
  Length 10           // Either "Length" or "Long Length" ( > 63 bytes)
  ShapeID 1
  RECT
    ...
  ShapeWithStyle      // all RGB becomes RGBA
    ...

// FrameLabel Tag
FrameLabel
  Tag 43              // Either "Length" or "Long Length" ( > 63 bytes)
  Length 10
  Name
    46                // F
    57                // W
    53                // S
    0                 // end of String

// DefineMorphShape Tag
DefineMorphShape
  Tag 46
  Length 10           // Either "Length" or "Long Length" ( > 63 bytes)
  CharacterID 1
  RECT                // Start Bounds
    ...
  RECT                // End Bounds
    ...
  Offset 30
  MorphFillStyles
    FillStyleCount 4    // "FillStyleCount" or " FillStyleCountExtended"
    FillStyleType 0     // 5 Different FillStyleTypes; 0 = solid fill
      RGBA              // Start Color
        Red 0
        Green 102
        Blue 204
        Alpha 100
      RGBA              // End Color
        Red 0
        Green 0
        Blue 0
        Alpha 0
    FillStyleType 16    // 16 = linear gradient fill
      Matrix            // Start Matrix
        ...
      Matrix            // End Matrix
        ...
      MorphGradient
        NumGradients 2 // Number of MorphGradRecords
```

```
      MorphGradRecord
        StartRatio 5
        RGBA
          Red 0
          Green 102
          Blue 204
          Alpha 100
        EndRatio 5
        RGBA
           Red 0
          Green 102
          Blue 204
          Alpha 100
      MorphGradRecord
        StartRatio 8
        RGBA
          Red 0
          Green 201
          Blue 102
          Alpha 80
        EndRatio 8
        RGBA
          Red 0
          Green 201
          Blue 102
          Alpha 80
FillStyleType 18  // 18 == radial gradient fill
  Matrix              // Start Matrix
    ...
  Matrix              // End Matrix
    ...
  MorphGradient
    NumGradients 2 // Number of MorphGradRecords
    MorphGradRecord
      StartRatio 5
      RGBA
        Red 0
        Green 102
        Blue 204
        Alpha 100
      EndRatio 5
      RGBA
         Red 0
        Green 102
        Blue 204
        Alpha 100
    MorphGradRecord
      StartRatio 8
      RGBA
        Red 0
        Green 201
        Blue 102
        Alpha 80
      EndRatio 8
      RGBA
        Red 0
        Green 201
        Blue 102
        Alpha 80
FillStyleType 64  // 64 == tilted bitmap fill
  BitmapID 2     // ID of bitmap character for fill
  Matrix         // StartBitmapMatrix
    ...
  Matrix         // EndBitmapMatrix
```

```
      ...
    FillStyleType 65  // 65 == tilted bitmap fill
      BitmapID 2    // ID of bitmap character for fill
      Matrix        // StartBitmapMatrix
        ...
      Matrix        // EndBitmapMatrix
        ...
  MorphLineStyles
    LineStyleCounts 2 // Either "LineStyleCount" or "LineStyleCountExtended"
    StartWidth 20
    EndWidth 20
    RGBA            // Start Color
      Red 0
      Green 201
      Blue 102
      Alpha 80
    RGBA            // End Color
      Red 0
      Green 201
      Blue 102
      Alpha 80
  Shape
    NumFillBits 12
    NumLineBits 12
    ShapeRecord
      ...           // in DefineShape Tag
  Shape
    NumFillBits 12
    NumLineBits 12
    ShapeRecord
      ...           // in DefineShape Tag

// ExportAssets Tag
ExportAssets
  Tag 56
  Length 10          // Either "Length" or "Long Length" ( > 63 bytes)
  Count 2
  Tag 100
  Name
    46              // F
    57              // W
    53              // S
    0               // end of String
  Tag 101
  Name
    46              // F
    57              // W
    53              // S
    0               // end of String
// Import Assets Tag
ImportAssets
  Tag 57
  Length 10          // Either "Length" or "Long Length" ( > 63 bytes)
  URL
    46              // F
    57              // W
    53              // S
    0               // end of String
  Count 2
  Tag 100
  Name
    46              // F
    57              // W
    53              // S
```

```
    0               // end of String
  Tag 101
  Name
    46              // F
    57              // W
    53              // S
    0               // end of String

// EnableDebugger Tag
EnableDebugger
  Tag 58
  Length 10         // Either "Length" or "Long Length" ( > 63 bytes)
  Password
    46              // F
    57              // W
    53              // S
    0               // end of String

// EnableDebugger2 Tag
EnableDebugger2
  Tag 64
  Length 10         // Either "Length" or "Long Length" ( > 63 bytes)
  Password
    46              // F
    57              // W
    53              // S
    0               // end of String
```

CHAPTER 6

Testing Plan

6.1 Objectives

No computer program can function appropriately without being tested layer after layer. It is common to only discover errors after the developed program has been thoroughly checked over by different testing plans. The objective of this section of the project is to achieve just that: analyze F2 from different angles and use several comprehensive approaches to test for errors at the various development phases. With careful choice of unit test, regression tests, white box and black box tests, we can guarantee a rather smooth evolution of program development.

6.2 General Concept

The general concept behind our testing plan is to design both F2 and Shockwave files as test programs that could help us, as the developers, to understand any loop holes that might have occurred during the actual implementation of our logical designs. These "helpers" can perform some general implementation checks as well as going deeper behind the design structure of F2. They are deployed against the F2 code to test smartly and efficiently.

6.3 Methods

To avoid prolong errors that could possibly effect the different structural parts of F2, our group seeks to run test programs in parallel throughout each of the development phases. These tests have been consistently modified throughout each phase to ensure testing upon program integrations. The Macromedia Flash Player is used as the main error checker for our sample F2 programs. The Flash player will display our .f2 files if these files are correct and recognizable by Macromedia Flash.

During the stage of grammar production, we decide to print out and evaluate the values of all nodes inside the AST tree to test the logical structure behind the grammar and the layout of our parser.

During the stage of backend and assembly development, we plan to use the Macromedia Flash User manual and the Macromedia Flash Player as the main

testing tools. This decision is based on the nature of SWF files. Their binary digit representation limits down available helper tools we can use.

For the code generation stage, we decide to use the developed backend programs, the established assembly language, and the parser as the main tools to run tests. The main testing objective at this stage is to guarantee the proper execution of our Movie methods in F2. Thus we plan to run source code through the parser and then perform code generation. Once this is done, we use our WriteSWF.java program to output the .f2 file into a .swf file and check the content with Flash Player.

6.5 Actual implementation

Grammar/Walker

To avoid compounding mistakes and error in our grammar, we choose to perform testing parallel to programming development. Upon the completion of each lexical rule, we build the necessary walker to proceed with many sample implementations. To display the hierarchy within these abstract syntax trees, we output all related tree nodes for logic checking. Only after all trials have been proven successful do we proceed again onto the next lexical rule.

After the lexer, the parser, and the walker have been completed, we begin testing the entire grammar by writing sample codes. We formulate different combinations of logical statements, object declarations, array declarations, Movie declaration calls to test against the grammar. By parsing out all the nodes and outputting them in the destined hierarchy order, we are able to check the logical structure and ensure the proper structure of the grammar.

Backend and Assembly Language

The initial testing runs for our backend are very tedious and labor intensive. Given that Flash files are stored in binary digits, the only way to check whether any parts of the assembly language has been correctly represented is to go back to the Macromedia Flash User Manual. The manual provides the actual binary representation of the specified tags. Once the Assembly language has been produced, testing has gone from the manual raw code checking to a higher level abstraction checking. Similarly with the building of backend programs such as ReadSWF.java and WriteSWF.java, the error within these programs could only be check by the Flash user manual. The main target in this testing phase is to ensure the correct binary recapitulation of Macromedia Flash tags within our intermediate assembly language and help programs from our backend.

Code Generation and Runtime Environment

We performed the testing at this stage mainly through using sample programs. To guarantee the correct inner working of methods developed inside this stage, we perform exactly the proposed methodology to do testing. We use different source codes to test against the different functions inside the Runtime Environment/Code Generation. We run sample source codes (.f2 files) through the parser, and then run Runtime Environment/code generation that output these .f2 files into a Shockwave file. Furthermore, we can check the correctness of the code through the Macromedia Flash Player. Any incorrect bit in the file will prevent the Flash Player from outputting the file. Thus it is very easy to know whether inner working of code generation is executing accordingly. Our last resort to check the methods are to produce copies of .swf files that are equivalent to what we want to present syntactically. By checking through the assembly language representation of both files (the Macromedia copy and our F2 file), we at times can find out what went wrong.

Comprehensive Testing for F2

At the completion of our F2 program development, we start the final testing for the entire F2 language. We begin with writing full versions of F2 programs that involves elaborate evaluation of Movie objects, logical statements, and a combination of methods defined in F2. We design source codes in such ways as to challenge the comprehensiveness of F2. And unfortunately, there are several minor design flows we cannot overcome in the given amount of time. First our proposed Composite Objects cannot be implemented properly. Second, our Text objects cannot be produced correctly as we planned. There are faulty cases. Moreover, a letter or number, with a given Font, is represented by 200 bytes on average. It is impossible to complete this Text library. Again, limited by time, we could not devise an appropriate algorithm for rotational movements.

Other than the above mentioned areas, all other aspects of F2 have been tested thoroughly. Results are positive. F2 executes as we have describe in the reference manual.

CHAPTER 7

Experience of F2 Language Development

The entire journey of F2 program development has been very rewarding. In a group environment, each of us learns how to utilize and prioritize things to achieve maximum efficiency and our overall objective. We realize that a good language development entails intensive structural planning and highly organized implementation details. Detail-oriented approach is very crucial during each phase of language development. Other factors such as comprehensive testing plans, thorough research on related material and proper time management are also integral factors toward a successful language development.

Other than the technical areas, we also learn how to work as a team. Computer language development has always required the joint force of several devoted individuals. The atmosphere between each individual team members has to be carefully maintained as to prevent any potential confrontation that could infringe the progress of the project.

Finally, we would like to say thanks to everyone who has helped us along the way. All the ideas, advices, and support we received throughout this project have contributed to the proper completion of the language development of F2. Thank you again.

APPENDIX A

F2 Grammar

WholeProgram:	(*Import-definition*) (*Header-statement*) (*Movie-block*) (*Frame-definition*)+ (EOF)
Constant:	*Int* | *String*
Type:	identifier | *Built-In-Type*
Built-In-Type:	Int | String | Boolean | void
Identifier:	ID (“.” ID)*
Field:	ID (*Parameter-Declaration-List*) (*Declarator-Brackets*) | *Variabel-declaration* “;”
Variable-Declarator:	ID (*Declarator-Brackets*) (*Variable-Initializer*)
Declarator-Brackets:	(“[“ “]”)*
Variable-Initializer:	(“=” *Initializer*)?
Array-Initializer:	“{“ (“,” *Initializer*)* (“,”)? “}”
Initializer:	*Expression* | *Array-Initializer*
Method-Head:	ID “(“ *Parameter-Declaration-List* “)”
Parameter-Declaration-List:	(*Parameter-Declaration* (“,” *Parameter-Declaration* ID)*)?
Parameter-Declaration:	ID *Declarator-Brackets*
Compound-Statement:	“{“ (*Statement*)* “}”
Statement:	*Compound-Statement* | *(Declaration)* | *Expression* “;” | “if” “(“ *Expression* “)” *Statement* (“else” *Statement*)? | “for” “(“ *For-Initializer* “;” *For-Condition* “;” *For-Iterator* “)” | “while” “(“ *Expression* “)” *Statement* | “break” (ID)? “;” | “return” (*Expression*)? “;” | “;”
For-Initializer:	(*Declaration* | *Expression-list*)?
For-Condition:	(*Expression*)?
For-Iterator:	(*Expression*)?

Expression:	*Assignment-expression*
Expression-list:	*Expression* (“ ,” *Expression*)*
Assignment-expression:	*Logic-Or-expression* | *Logic-Or-expression* (“ =” *Assignment-expression*)
Equality-expression:	*Relational-expression* ((“!=” | “==”) *Relational-expression*)*
Relational-expression:	*Additive-expression* ((“<” | ”>” | “<=” | “>=”) *Additive-expression*)*
Additive-expression:	*Multiplicative-expression* ((“+” | “-“) *Multiplicative-expression*)*
Multiplicative-expression:	*Unary-expression* ((“*” | “/” | “%”) *Unary-expression*)*
Primary-expression:	*Identifier-primary* | *constant* | *“true”* | *”false”* | *“null”* | *New-expression* | (*Assignment-expression)*
New-expression:	“new” type *NewArrayDeclarator* (*Array-initializer*)?
NewArrayDeclarator:	[(*expression*)?] | *NewArrayDeclarator+*
ParameterDeclaration:	*AtomicObjectType identifier* | String *identifier* | int *identifier*
AtomicObjectType:	Text | Rectangle | Circle | Ellipse | Line
ImportDeclaration:	import *StringExpression* ;
MovieDeclaration:	MOVIENAME = *StringLiteral* ;
FrameWidthDeclaration:	WIDTH = *IntExpression* ;
FrameHeightDeclaration:	HEIGHT = *IntExpression* ;
FPSDeclaration:	FPS = *IntExpression* ;
NumFramesDeclaration:	NUMFRAMES = *IntExpression* ;
BGColorDeclaration:	BGCOLOR = *ColorExpression* ;
ColorExpression:	BLACK | BLUE | BROWN | GREEN | GREY | ORANGE | PURPLE | RED | WHITE | YELLOW
Movie:	Movie { *Statement* }
F2Declaration:	*AtomicObjectDeclaration*

| Composite*ObjectDeclaration*
| *Declaration*

AtomicObjectDeclaration: Text *identifier* (*TextParam*) ;
| Rectangle *identifier* (*RectangleParam*) ;
| Circle *identifier* (*CircleParam*) ;
| Ellipse *identifier* (*EllipseParam*) ;
| Line *identifier* (*LineParam*) ;

TextParam: *StringExpression* , *IntExpression* , *ColorExpression*

RectangleParam: *IntExpression* , *IntExpression* , *ColorExpression*

CircleParam: *IntExpression* , *ColorExpression*

EllipseParam: *IntExpression* , *IntExpression* , *ColorExpression*

LineParam: *IntExpression* , *IntExpression* , *IntExpression* ,
IntExpression , *ColorExpression*

AnonAtomicObjectDeclarationList:
AnonAtomicObjectDeclaration
| *AnonAtomicObjectDeclarationList*
| *AnonAtomicObjectDeclaration*

AppendStatementList: *AppendStatement*
AppendStatementList , *AppendStatement*

AppendStatement: Append (*identifier* , *IntExpression* , *IntExpression*) ;

FrameDefinition: Frame (*IntExpression*) { *StatementList* }

PlaceDeclaration: Place (*StringLiteral* , *IntExpression* ,
IntExpression , *IntExpression* , *InteExpression* ,
IntExpression) ;

AnimateDeclaration: Animate (*StringLiteral* , *IntExpression* ,
IntExpression , *IntExpression* , *IntExpression* ,
IntExpression , *IntExpression* , *IntExpression*) ;

InsertDeclaration: Insert (*StringLiteral* , *IntExpression* ,
IntExpression , *IntExpression* , *IntExpression*) ;

APPENDIX B

Code Style Conventions

B.1 Introduction

F2 integrates the java code syntax along with its own distinct set of syntax to form a new grammar appropriate for coding macromedia definitions. The intention behind this document is to recommend to the user a uniform standard for collaborative program development. In general, every programmer has his or her own specific coding style, such as how he or she places curly braces, semi-colons and cases. The objective of this document is not to force users to follow a specific coding style preferred by the designers of F2, but rather, to present a general coding convention that reflects clarity and ease for maintenance.

B.2 General Format

Just as recommended by every existing programming language in the world, it is a good practice to produce code that is easy to read and sensibly laid out. Variables should be named accordingly to reflect their purpose and commented as users see fit. Blocks should be indented with a consistent width for good formatting; nested blocks should be indented appropriately to reflect hierarchy of execution. It is good convention to provide explanatory comments for all nontrivial blocks.

B.3 Documentation Comments

By convention, every public method and field of Java class should be documented with a Javadoc comment. For F2, it is recommended to perform such measure to clarify the meaning and structure of a F2 program. Comments on local methods should describe formal parameters and explain their functionalities. The specification of F2 objects should be thoroughly covered in comments. Moreover, given that F2 produces macromedia swf files, it is imperative to include, at the beginning of the program, the complete synopsis of the desired macromedia file.

Bibliography

- Macromedia Flash (SWF) File Specification Version 6
- Antler Reference Manual
- The C Programming Language by Kernighan and Ritchie
- Java Software Solutions By Lewis Loftus
- Java API by Suns Microsystems