Programming Languages and Translators
COMS W4115
Stephen A. Edwards
Fall 2007

Orange White Green Animation Language
Reference Manual

Meenakshi Sripal
October 18, 2007

Language Reference Manual

1.1. Introduction

OWGAL language is a programming language that allows users to develop animations through the generation of shockwave flash files. This manual describes in details about the rules, conventions, and built-in features.

1.2. Grammar Notation

This manual introduces lexical and syntactic aspects of OWGAL's grammar. A set of productions are defined consisting of both non-terminals and terminals. The lexical grammar has undefined terminals such as if keyword, symbols like ( )[ ], else keyword; terminals are given in arial font. Examples of non-terminals are *expression* or *identifier*; non-terminals are given in *italic* font. A non-terminal are always defined on the left side of the colon whereas a mix of terminals and non-terminals are defined on the right side of the same colon. The syntactic grammar describes how sequences of tokens can form syntactically correct programs.

The input sequence of characters of the program are scanned and grouped into tokens, in which white space and comments discarded. If the sequence of characters is in the form *a*?, it denotes that the symbol *a* is optional. *a** denotes that the symbol *a* may occur zero or more times. *a*+ denotes that the symbol *a* will occur one or more times. (*a*|*b*) denotes a choice between the symbols *a* and *b*.

1.3. Lexical Conventions

In order for the OWGAL program to run, an object, mainly an image, can be imported into the code. Compiling the animation file will generate a syntactically correct program, which in turn can be used by the Java compiler. The goal of the lexical grammar is to simplify the job of the parser, which sees the animation file prior the Java compiler.

1.3.1. Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. White space, such as blanks, tabs, newlines, and comments, are ignored except as they are used to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

1.3.2. Comments

There are two kinds of comments:

```
! text !
```

The characters ! introduce a comment, which terminates with the character !.

```
!! text
```

Single-line comments introduced by !! (double exclamation point) cause the compiler to ignore the remainder of that line.

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

1.3.3. Identifiers

An *identifier* is a sequence of letters and digits. It must begin with a letter. Identifiers are case sensitive and can have any length. The underscore character (_) is interpreted as a letter.

1.3.4. Keywords

The following identifiers are reserved for use and cannot be used as otherwise:

text	width	if	ROUTINE
for	length	else	return
import	img	xposition	yposition
background	foreground	set	put
while	move	up	down
left	right	xaxis	yaxis

1.3.5. Constants

There are several kinds of constants.

constant: *integer-constant*
character-constant

1.3.5.1. Integer Constants

An integer constant consists of a sequence of decimal digits ranging from 0 to 9. It is unsigned and always positive. The maximum value for the integer constant is 2^{32}.

1.3.5.2. Character Constants

A character constant is a sequence of one or more characters enclosed in double quotes (“…”). Character constants do not contain the “ character or newlines; in order to represent them, the following escape sequences may be used.

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

1.4. Meaning of Identifiers

Identifiers, or names, can 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.

1.4.1. Basic Data Types

Objects declared as text contain a sequence of characters. Objects declared as img contains special file format and attributes that are recognizable by the compiler.

1.5. 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.

1.5.1. Primary Expressions

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

```
        primary-expression:   identifier
                              | constant
                              | (assignment-expression)
```

An identifier is a primary expression. Its type is specified by its declaration.

A constant is a primary expression, with either form of the following types: text, img, length and width.

A parenthesized expression is a primary expression containing a mix of nonterminals and terminals.

1.5.2. Additive Operators

The additive operators + and – are operated from left to right. If the operands have arithmetic type, the usual arithmetic conversions are performed.

```
additive-expression:   integer-expression + atom
                       | integer-expression – atom
                       | atom ;
atom: NUMBER ;
```

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

1.5.3. Relational Operators

The relational operators are operated from left to right.

relational-expression:	*relational-expression* < *additive-expression*
	| *relational-expression* > *additive-expression*
	| *relational-expression* <= *additive-expression*
	| *relational-expression* >= *additive-expression*

1.5.4. Equality Operators

equality-expression:	r*elational-expression*
	| e*quality-expression* == r*elational-expression*
	| e*quality-expression* != r*elational-expression*

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence.

1.5.5. Assignment Expressions

assignment-expression:	p*rimary-expression*
	| i*dentifier* = a*ssignment-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 i*dentifier*.

1.6. Declarations

declaration:	*TypeSpecifier InitIdentifierList ;*
type-specifier:	text
	| img
init-identifier-list:	*init-identifier*
	| *init-identifer-list, init-identifier*
init-identifer:	*identifer*
	| *identifier = string-expression ;*

A declaration consists of a *type-specifier,* followed by an *identifier,* and possibly followed by an equal sign and a s*tring-expression* or an i*nteger-expression* (if the user chooses to declare and initialize).

1.6.1. Function Declarations and Definitions

Functions are declared and defined outside of the ROUTINE block. They are type-specified by a text. Following the *type-specifier* is the *identifier*, and following the *identifier* is the parameter list surrounded by parentheses.
The syntax of the parameters is

parameter-type-list:	*parameter-list*
	| *parameter-list,…*
parameter-list:	*parameter-declaration*

	| *parameter-list, parameter-declaration*
parameter-declaration:	text *identifier*
	| img *identifier*

The function is then defined within a pair of braces. At the end of function definition, the function returns a string.

1.7. Statements

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

statement:	*expression-statement*
	| *compound-statement*
	| *conditional-statement*
	| *iteration-statement*

1.7.1. Expression Statement

Most statements are expression statements, which have the form:

expression-statement: *(expression)* ;*

Most expression statements are assignments.

1.7.2. Compound Statement

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement: { (declaration-list) (statement-list)* }*

declaration-list:	*declaration*
	| *declaration-list declaration*

statement-list:	*statement*
	| *statement statement-list*

An identifier within the declaration-list may be declared only once in the same block.

1.7.3. Conditional Statements

Conditional statements choose one of several flows of control.

conditional-statement:	if (*expression*) *statement*
	| if (*expression*) *statement* else *statement*

In both forms of the if statements, the expression is evaluated and the first sub-statement is executed only if the expression is not equal to zero. In the second form, the second sub-statement is executed only if the expression is zero. The else ambiguity is resolved by connecting an else with the last encountered else-less if at the same block nesting level.

1.7.4. Iteration Statements

Iteration statements specify looping:

iteration-statement:	while (*expression*) *statement*
	| for (*expression ; expression ; expression)* *statement*

In the while statement, the sub-statement is executed repeatedly so long as the value of the expression is not equal to zero; the expression must have 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 its type. The second expression must have arithmetic; it is evaluated before each iteration, and if it becomes equal to zero, 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.

1.8. OWGAL Source File Specifications

This section describes the format of an OWGAL source file.

1.8.1. ROUTINE Block Definition

Routine:	ROUTINE { *CompoundStatement* }
CompoundStatement:	*DeclarationList StatementList*
StatementList:	*Statement*
	| *StatementList Statement*
DeclarationList:	*Declaration*
	| *DeclarationList Declaration*
Declaration:	*ImageDeclaration*
	| *(PrimitiveTypeDeclaration)**
ImageDeclaration:	*(ImportDeclaration)+*
PrimitiveTypeDeclaration:	text *IdentifierList ;*
	| img *IdentifierList ;*
IdentifierList:	*Identifier*
	| *IdentifierList, Identifier*

The ROUTINE block definition is similar to the "main" routine declaration in a C program. The declarations and statements must be made within the ROUTINE block, and nothing should follow after the closing curly brace.

1.8.2. Image Declaration Statements

ImageDeclaration: img *identifier = ImportDeclaration ;*

ImageLengthDeclaration: *identifier*.length = *IntegerExpression ;*

ImageWidthDeclaration: *identifier*.width = *IntegerExpression ;*

The image declaration statements allow users to set up the primary objects that will be used throughout the animation. *ImageDeclaration* should be declared before declaring the image's length and width. There must be at least 1 image declaration statement within the ROUTINE block. The length and the width declarations are optional. If both of these are not declared, then the image's length and width will default to 50, 50, respectively. They all must be declared within the ROUTINE block.

img defines a special file format that needs to be recognized by the compiler.
length defines the length of the image in pixels. The length value must be a positive integer.
width defines the width of the image in pixels. The width value must be a positive integer.

1.8.3. Import Statement

ImportDeclaration: `import[` *FilenameExpression];*

The *`import`* statement tells the compiler to prepare a data structure based upon another program for use by the current program and load the image onto the screen during the execution. At least one import statement must always be specified at the very beginning of the program.

The *`import`* feature provides users with ease of use and flexibility by permitting users to input the already made images, with file extensions .jpeg or .gif, into the program to work with. Images can be photos or graphics. Checks will verify that the imported image file meets the file requirements.

1.8.4. Put Declaration

PutDeclaration: put *[identifier, IntegerExpression, IntegerExpression];*

The put parameters are: object name, x coordinate, y coordinate.

The put declaration is used to place objects (images) at the specified location permanently. The object name must be taken from the identifier in the

ImageDeclaration. The integers must be positive. There must be at least one image declaration before the put declaration can be specified. The put declaration must be declared within the ROUTINE block.

1.8.5. Set Declaration

SetDeclaration:	set *identifier BackFrontExpression ;*
BackFrontExpression:	background
	| foreground ;

The set parameters are: object name, background/foreground option.

The background option is used to place the specified object (image) behind other objects, if any. The foreground option is used to place the specified object in front of any images that are set to the background layer. If set is not declared, then the images will default to the front layer and images may overlay. There must be at least one image declaration before the set declaration can be specified. The set declaration must be declared within the ROUTINE block.

1.8.6. Move Declaration

MoveDeclaration:	move (*identifer, XYExpression, IntegerExpression, DirectionExpression) ;*
XYExpression:	xaxis | yaxis
DirectionExpression:	up | down | left | right

The move parameters are: object name, x/y axis, number of times, direction for the object to move.

The move declaration lets the user to dictate how the object (image) to be moved in a certain way. Integers must be positive. There must be at least one image declaration before the move declaration can be specified. The move declaration must be declared within the ROUTINE block.