Animo Reference Manual

Natasha Shamis (ns2104), Joshua Poritz (jsp2104),
Alexei Masterov (am2268)
Columbia University in the city of New York
October 19th, 2005

1. Introduction

This manual describes the first draft of the ANIMO language as decided by the ANIMO team on the Sixteenth of October, 2005. The manual serves as the authoritative standard for the ANIMO language and delineates all of ANIMO's rules, conventions, and built-in features. The standard libraries provided with ANIMO will be documented elsewhere in the near future.

2. Lexical Conventions

A program consists of one or more *translation units* stored in files. It is translated in two phases. The first phase attempts to interpolate (recursively) all files specified in `include` statements. These interpolated files may be ANIMO programs or "still BVH" (hereafter abbreviated SVBH) files that the program will animate. The second phase compiles the single resulting file into an animated BVH file, which is written to disk.

For a formal description of the SBVH format, see (http://www.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html). The rest of this section describes the tokens that comprise an ANIMO source file.

2.1. Tokens

There are six classes of tokens: identifiers, keywords and built-in functions, constants, string literals, operators, and other separators. Blanks, tabs, carriage returns, newlines, and comments as described below (collectively "white space") are ignored except as separate tokens. Some white space is required to separate adjacent identifiers, keywords, and constants.

2.2. Comments

ANIMO allows two styles of comments. Single-line comments, which are introduced by a double-slash ("//"), cause the compiler to ignore the remainder of that line. Multi-line comments begin with a slash followed by a star ("/*") and end with the first subsequent instance of a star followed by a slash ("*/"); the compiler ignores all text in between those two delimiters. Comments do not nest, and they do not occur within string literals.

2.3. Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Identifiers are case sensitive. All variables, BVH joint names, and function names fall under the classification of identifier.

2.4. Keywords and Built-in Functions

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
else        include     return
for         func        to
if
```

The following identifiers are reserved as names for built-in functions, and no user-defined function may have these names:

```
move        rotate
```

2.5. Constants

Almost exclusively, the only constant used throughout an ANIMO program is the floating point. A floating constant consists of an integer part, a decimal part, a fraction part, an e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

The only other constant employed in an ANIMO program is the string literal, which has its place only as the parameter to an `include` statement. A string literal consists of a sequence of characters enclosed in double quotes, as in `"... "`. Consecutive pairs of double quotes within a string literal are reduced to one double quote and that double quote is interpreted as part of the string literal, not as its end delimiter.

3. Expressions

This section places each type of expression operator in a separate subsection in order of precedence, from highest to lowest. The operators within each subsection have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

3.1. Primary Expressions

Primary expressions group left-to-right, and include identifiers, constants, parenthesized expressions, and function calls. The type and value of a parenthesized expression are identical to those of the simple expression. The presence of parentheses does not affect whether the expression is an l_value. A function call is composed of a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to the function.

3.2. Unary Operators

The unary operators, +, -, ++, --, and ! group right-to-left. They yield the positive, negative, increment, decrement, and negation of the expressions, correspondingly.

3.3. Multiplicative Operators

The multiplicative operators *, /, and % group left-to-right. The binary * operator indicates multiplication. The binary / operator indicates division. The binary % operator yields the remainder from the division of the first expression by the second.

3.4. Additive Operators

The additive operators + and – group left-to-right. The + operator yields the sum of the expressions, and the – operator yields the difference of the expressions.

3.5. Relational and Equality Operators

The relational operators <, >, <=, and >= group left-to-right. The equality operators == and != can be grouped in either direction. Both the relational and the equality operators yield 0 if the specified relation is false and 1 if it is true.

3.6. && Operator

The && operator groups left-to-right. It returns 1 if both expressions are non-zero, 0 otherwise. If the first expression evaluates to 0, the second expression is not evaluated.

3.7. || Operator

The || operator groups left-to-right. It returns 1 if either of its expressions is non-zero, 0 otherwise. If the first expression is non-zero, the second expression is not evaluated.

3.8. Assignment Operators

The assignment operators =, +=, -=, *=, /=, and %= group right-to-left. They require an l_value as their left operand, and the type of an assignment expression is the same as the left operand. After the assignment takes place, the resulting value is stored in the left operand.

4. Functions

A function encapsulates a sequence of ANIMO statements into a named piece of code. All functions in an ANIMO source program consist of a declaration, followed immediately by zero or more statements enclosed in curly braces. A declaration has the form:

func_def:
"func" *identifier* "(" *var_list* ? ")"
var_list:
identifier ("," *identifier*)*

The *identifier* in a *func_def* is the most heavily restricted type of identifier in an ANIMO program. The identifier must neither clash with any built-in keywords nor with any built-in function names. See section 2.4 for a complete list of built-ins.

The return value of a function is always a floating point constant. If no specified value is returned from within the function, the return value defaults to 0.

5. Statements

Except as indicated, statements are executed in sequence.

5.1 Expression statement

Most statements are expression statements, which have the form
expression ;

Usually expression statements are assignments or function calls.

5.2 Compound statement

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

compound-statement:
{ statementlist }

statementlist:
statement
statement statement-list

5.3 Conditional statement

The two forms of the conditional statement are

`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

In both cases the expression is evaluated and if it is nonzero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an else with the last encountered elseless if.

5.4 For statement

The for statement has the form

for (*id ASGN NUMBER* "to" *NUMBER*) *statement*

5.5 Return statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return ( expression ) ;
```

In the first case 0 is returned. In the second case, the value of the expression is returned to the caller of the function. Flowing off the end of a function is equivalent to "`return ;`".

5.6 Include Statement

Include statement is used to import other files, such as external libraries. We also allow for the use of include to import a Still BVH file (Bvh file that describes the skeleton, but not it's motion) Note: Only one skeleton can be imported per program. The skeleton must be imported before any statements are executed.

Include has the following format:

```
Include <filename> ;
```

If the filename ends with ".sbvh" it will call the BVH parser, otherwise the external file will be inserted in place of include during the first pass.

6. Scope

An ANIMO program has two levels of scope: Global scope, and block scope. An identifier initialized outside of any block (a group of statements enclosed in curly braces) falls under global scope and is visible throughout the remainder of the source file. By contrast, an identifier initialized within a block is restricted to block scope and is visible only for the remainder of the block in which it is declared.

BVH joint names are the exception to normal scoping rules. Any BVH joint name, including those declared within curly braces (i.e. not at the very top of the joint hierarchy), automatically has global scope and may be passed to any function for the remainder of the source file after which it appears.

As expected, function identifiers have global scope, whereas parameters specified in variable lists within function declarations have block scope.

Variable names, function names, and BVH joint names each comprise separate namespaces. The same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces.

7. Grammar

Below is a recapitulation of the grammar that has been heretofore described. Words in *italics* denote nonterminal symbols, whereas quoted words and symbols in `typewriter` font specify terminals. All other characters constitute regular expression operators and their meanings are fully described in Chapter 3 of Aho, *Compilers* and the documentation for ANTLR, a compiler generator. A slight modification of this grammar can be fed through to ANTLR to produce an ANIMO compiler.

program:
(*statement* | *func_def*)*

statement:
for_stmt | *if_stmt* | *return_stmt* | *load_stmt* | *assignment* | *func_call_stmt* |
"{" (*statement*)* "}"

for_stmt:

“for” “(” *identifier* “=” *number* “to” *number* “)” *statement*

if_stmt:
“if” “(” *expression* “)” *statement* (“else” *statement*)?

return_stmt:
“return” (*expression*)? “;”

load_stmt:
“include” *string* “;”

assignment:
l_value (“=” | “+=” | “-=” | “*=” | “/=” | “%=”) *expression* “;”

func_call_stmt:
func_call “;”

func_call:
identifier “(” *expr_list* “)”

expr_list:
expression (“,” *expression*)*

func_def:
“func” *identifier* “(” *var_list* ? “)” *func_body*

var_list:
identifier (“,” *identifier*)*

func_body:
“{” (*statement*)* “}”

expression:
logic_term (“||” *logic_term*)*

logic_term:
relat_expr (“&&” *relat_expr*)*

relat_expr:
arith_expr ((“>=” | “<=” | “>” | “<” | “==” | “!=”) *arith_expr*)?

arith_term:
arith_term ((“+” | “-”) *arith_term*)*

arith_term:
arith_factor ((“*” | “/” | “%”) *arith_factor*)*

arith_factor:
(“+” *r_value*) | (“-” *r_value*) | (“++” *r_value*) | (“--” *r_value*) | (“!” *r_value*) |
r_value

r_value:

l_value | *func_call* | *number* | “(” *expression* “)”

l_value:
identifier

identifier:
alpha (*alpha* | *digit*)*

number:
(*digit*)+ (“.” (*digit*)*)? ((“E” | “e”) (“+” | “-”) (*digit*)+)?

string:
“"” (~(“"” | “\n”) | (“"” “"”))* “"”

alpha:
“a” .. “z” | “A” .. “Z” | “_”

digit:
“0” .. “9”