Sprite

an animation manipulation language
Language Reference Manual

Team Leader
Dave Smith

Team Members
Dan Benamy
John Morales
Monica Ranadive

Table of Contents

A. Introduction

This language reference manual outlines the Sprite programming language. We used regular expression notation to describe the language. This manual will use the following notation within the regular expressions:

( ) – the enclosed terminals and nonterminals are a group

'a' – the symbol 'a' is a terminal

a? – the symbol 'a' is optional

a* – the symbol 'a' will occur, zero, one, or more times

a+ – the symbol 'a' will occur, one or more times

a | b – a choice between the symbols 'a' and 'b'

Syntactic categories are in *italic* type, and literal words and characters are in `typewriter` style.

B. Lexical Conventions

A program consists of one Sprite specification stored in a file. Programs are in the ASCII character set.

B1. Tokens

The language contains five classes of tokens: identifiers, keywords, constants, string literals, and operators. Blank space, any type of tab, and comments are classified as “white space” and are ignored by the parser.

B2. White Space

A line break denotes the end of a statement. Lines of programming may not wrap. Spaces, horizontal tabs, and carriage return characters are whitespace. Comments are ignored by the tokenizer. Sprite occasionally requires whitespace to separate adjacent tokens that would otherwise be a single token. For example, identifiers and keywords must have separating whitespace. The newline character at the end of a file is optional.

B3. Comments

Single line comments start with `<<` and end at a carriage return (the only line terminator available) or at the end of file. No special characters denote multiple line comments.

B4. Identifiers

An identifier is a sequence of letters or digits, the first of which much be a letter. Letters are considered any letter of the alphabet (upper and lower case) as well as '_'. Upper and lower case letters are different. Two identifiers are the same if they have the same ASCII character for every letter and digit.

Identifier → *letter (letter | digit)**

B4.1 Primitive Data Types

The primitive data types for identifiers are strings, numbers, and boolean values (true/false). Sprite does not require type declarations. Sprite implicitly casts to strings in certain situations, see §B9.

B4.2 Objects

Objects may contain other objects and primitive data types. Objects specify defaults for all primitive data types. Objects will not contain constructors or subroutines.

```
obj foo
{
    bar = 1
    rab = true
    arb = "Hello world"
}
```

B4.3 Arrays

The `add(`*array_identifier*`,` *identifier*`)` function may append any primitive or object to an array. The `count()` function returns the number of elements in an array. Access the nth element of an array using *array_identifier*[*n – 1*]. Arrays manage their own memory, and allocate space as more elements are added.

Array out of bounds errors will return one of two special values: undefined and NaN (not a number). These error occur at runtime. Array out of bounds errors return undefined unless it is part of a multiplicative/additive expression, and will return NaN.

B5. Keywords

Sprite reserves the following identifies as keywords; all keywords are lower case and use only letters.

`add`	`elseif`	`print`
`array`	`false`	`sprite`
`callevery`	`for`	`ret`
`count`	`func`	`true`
`if`	`new`	
`else`	`obj`	

B6. String Literals

A string literal, or string constant, is a sequence of characters surrounded by double quotes, like " ... ". The addition operator concatenates strings. String literals may not contain double quotes or span multiple lines.

B7. Separators

The following characters are separators:

```
{ } , . [ ]
```

Curly brackets {} must be on their own line and cannot accompany any other code on the line on which they appear. Sprite does not allow arbitrarily placed separators. See §D.

NOTE: This document, in order to save space, places curly brackets on the same line as code. This will not compile correctly, as curly brackets must be on their own line.

B8. Constants

Constants are a sequence of digits representing an integer or real. If an arithmetic expression between two integers evaluates to a real, the output will be automatically cast as a real.

Number → *digit+* (`.` *digit+*)?

B9. Type casting

Addition of a String with any primitive non-String implicitly type casts the non-String to a String. The `print(`*expression*`)` function implicitly type casts any non-String argument as a String.

B10. Scope

Static scoping rules in Sprite are as follows:

1. Identifiers are valid from where they are initialized and are no longer valid after the end of the block within which they were made.
2. Identifiers accessible where a new block starts are valid inside that block.
3. A file that does not contain any {}'s is considered one block for scoping purposes.

B11. Garbage Collection

Sprite has built-in garbage collection. Dynamically allocated instances are automatically freed at some point after the last reference to them is destroyed.

C. Expressions

The precedence of expression operators is identical to the order in which they are documented below. Within each subsection, the operators have the same precedence. Each subsection is left-associative unless otherwise specified. The order of evaluation of expressions is left to right.

Expressions are the building blocks of statements and programs in the Sprite Language.

C1. Primary Expressions

factor:
| (*expression*)
| *left-value*
| *number*
| *boolean*
| *string*

C2. Arithmetic Expressions

Arithmetic expressions take the primary expressions as their operands. While the productions of the expressions appear to create right associative rules, when implemented in ANTLR, the language is left-associative.

C2.1 Unary Expressions

unary:
| – *unary*
| ! *unary*
| *factor*

A unary expression is an operation with at most one operand ('-' or '!'). The prefix notation binary operators '-' and '!' represent negative and not.

C2.2 Multiplicative Expressions

mult-expression:
| *unary* (* *unary*)*
| *unary* (/ *unary*)*

The binary operators '*' and '/' represent multiplication and division.

C2.3 Additive Expressions

add-expression:
| *mult-expression* (+ *mult-expression*)*
| *mult-expression* (– *mult-expression*)*

The binary operators '+' and '-' indicate addition and subtraction.

C2.4 Comparison Expressions

comp-expression:
| *add-expression* (> *add-expression*)*
| *add-expression* (< *add-expression*)*
| *add-expression* (>= *add-expression*)*
| *add-expression* (<= *add-expression*)*

equal-expression:
| *comp-expression* (== *comp-expression*)*
| *comp-expression* (!= *comp-expression*)*

join-expression:
| *equal-expression* (&& *equal-expression*)*

expression:
| *join-expression* (|| *join-expression*)*

Comparison expressions can intuitively compare primary expressions. Using the Kleene close an expression can evaluate to any combination of comparisons and mathematical expressions or to a simple terminal such as a number of string.

C3. Assignment Expressions

To assign an argument to a value, a declaration must be made as detailed below:

assignment:
| *left-value* = *right-value*

Primitives are assigned by value. Objects and Arrays are assigned by reference.

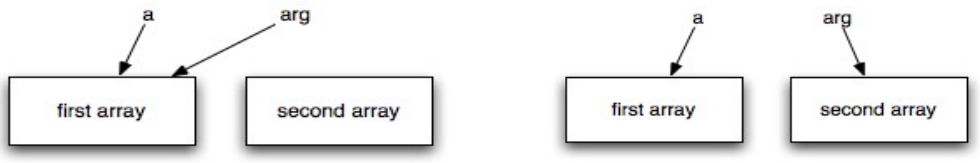
Example 1

```
a = new array
b = a
add(b,5)
```

In this example at the end, both a and b still have the same value. There is only one array, both a and b point to it.

Example 2

```
func foo(arg)
{
    << arg = new array
    add (arg, 2)
}
a = new array
add (a, 5)
foo(a)
```



With the comment left in, the above code will evaluate to the left figure. If the line is uncommented, arg will point to a new Array, as shown in the right figure.

C3.1 Left-Value Expressions

left-value:
| *identifier* | *array* | *member*
array:
| *identifier* (`[` *expression* `]`)+
member:
| *identifier* (`.` *identifier*)+

A left-value expression describes all valid syntax that may appear on the left hand side of an assignment. This is used for declaring/accessing identifiers, arrays, and characteristics of identifiers.

C3.2 Right-Value Expressions

right-value:
| `new array`
| `new` *identifier*
| *expression*
| *function*

A right-value expression describes all valid syntax that may appear on the right hand size of an assignment. This is used for declaring and accessing arrays, identifies, expressions, and functions.

D. Sprite Program

D1. Programs

Sprite programs are a sequence of one or more statements.

program:
| *statement* +

Certain types of statements may contain blocks of code made up of more statements which will be formatted as follows:

block:
| `{` *statement* * `}`

D2. Statements

Statements are executed in sequence. Statements can fall into several groups.

statement:
| *function*
| *selection-statement*
| *add-statement*
| *count-statement*
| *iteration-statement*
| *func-statement*
| *object-statement*
| *print-statement*
| *sprite-statement*
| *ret-statement*

D2.1 Functions

Functions are created using a function-statement, with a name (their identifier) as well as a list of parameters (expressions) which the function will take upon declaration. Function may except zero or more parameters, each separated with a comma. The function statement refers to the correct grammar for calling a function which has been previously declared.

function:
| *identifier* `(` *(expression (, expression)*)?* `)`

D2.2 Selection Statements

Selection statements must be formatted as follows.

selection-statement:
| `if` *(* *expression)*
block
(`elseif (`*expression* `)`
*block)**
(`else`
block)?

In both forms of the `if` statement, the *expression* is evaluated and if it evaluates to `true` the *block* on the following line is executed. In the second form, the second *block* is executed if the *expression* evaluates to false. There is no else ambiguity because of the rule that { }'s must surround a block.

D2.3 Add Statements

Add statements allow elements to be added to arrays.

add-statement:
| `add (` *identifier , expression* `)`

An add statement consists of an *identifier* representing an array and an *expression* representing the element to be added to the array. Arrays are heterogeneous so it does not matter what type expression evaluates to.

D2.4 Count Statements

Count statements will evaluate the number of elements in a given array.

count-statement:
| `count (` *identifier* `)`

The *identifier* represents an array and the statement will evaluate to the number of elements held in the array specified.

D2.5 Iteration Statements

Iteration statements specify looping.

iteration-statement:
| `for (` *assignment* `;` *expression* `;` *assignment* `)`
block

Unlike other languages, the arguments within the *for* statement are required for the loop to compile correctly. In the `for` statement, the first assignment is evaluated once, and specifies the initialization for the loop. The *block* must start on a new line of code, comments are the only thing allowed on the same row as a `for` statement.

D2.6 Function Statements

func-statement:
| `func` *function callEvery-exp?*
block
callEvery-exp:
| `callevery (` *number* | *left-value* `)`

Functions executes a block of sprite code with optional arguments, passed by reference. A function may include a callevery option. The callevery(*int-n*) option causes the function to be repeatedly executed every *int-n* milliseconds.

D2.7 Object Statements

There is only one way to write an object statement. Object statements are *object* declarations in Sprite.

object-statement:
| `obj` *identifier* `{` *(assignment \n)* `+` `}`

D2.8 Print Statements

The print statement displays data.

print-statement:
| `print (` *expression* `)`

If the statement is passed something other than a string, the terminal of expression will be cast to a String.

D2.9 Sprite Statements

Sprite statements are obvious a particularly important section of the language.

sprite-statement:
| `sprite (` *string*|*member* `,` *constant*|*member* `,`
constant|*member* `,` *constant*|*member* `)`

A sprite statement accepts a string/member (the name of the .jpg file) as well as three additional parameters that evaluate to numbers. The numbers refer to the coordinates where the image file will be displayed.

D2.10 Return Statements

A return statement is used within a function to return a particular right-value.

ret-statement:
| `ret` *r-value*

E. Code Sample

The sample of code describes a single sprite, ball, bouncing around the screen of a browser window.

```
obj Ball
{
     left = 50
     top = 50
     downDirection = 3
     rightDirection = 3
     id = 0
}

newBall = new Ball
newBall.id = 1
func MoveBall () callevery(10)
{
     newBall.top = newBall.top + newBall.downDirection
     newBall.left = newBall.left + newBall.rightDirection
     if (newBall.top <= 0 || newBall.top >= 400)
     {
          newBall.downDirection = -1 * newBall.downDirection
     }
     if (newBall.left <= 0 || newBall.left >= 700)
     {
          newBall.rightDirection = -1 * newBall.rightDirection
     }
     sprite("Ball.jpg", newBall.id, newBall.top, newBall.left)
}
```