# *Euclides* – A JavaScript to PostScript Translator

Martin Strobl, Christoph Schinko
*Institut für ComputerGraphik und WissensVisualisierung*
*Technische Universität Graz, Austria*
{ *m.strobl, c.schinko* }*@cgv.tugraz.at*

Torsten Ullrich[1], Dieter W. Fellner[2]
[1,2] *Fraunhofer Austria, Graz, Austria*
[2] *Fraunhofer IGD & TU Darmstadt, Germany*
*torsten.ullrich@fraunhofer.at*
*d.fellner@igd.fraunhofer.de*

***Abstract*—Offering an easy access to programming languages that are difficult to approach directly dramatically reduces the inhibition threshold. The Generative Modeling Language is such a language and can be described as being similar to Adobe's PostScript. A major drawback of all PostScript dialects is their unintuitive reverse Polish notation, which makes both - reading and writing - a cumbersome task. A language should offer a structured and intuitive syntax in order to increase efficiency and avoid frustration during the creation of code. To overcome this issue, we present a new approach to translate JavaScript code to GML automatically. While this translation is basically a simple infix-to-postfix notation rewrite for mathematical expressions, the correct translation of control flow structures is a non-trivial task, due to the fact that there is no concept of "goto" in the PostScript language and its dialects. The main contribution of this work is the complete translation of JavaScript into a PostScript dialect including all control flow statements. To the best of our knowledge, this is the first *complete* translator.**

***Keywords*-PostScript; JavaScript; translator; transpiler**

## I. MOTIVATION

The language PostScript [1] by JOHN WARNOCK and CHARLES GESCHKE at Adobe Systems is a dynamically typed concatenative programming language which is known for its use as a page description language for desktop publishing. Beginning in the 1980s PostScript (PS) and its descendants, namely the Portable Document Format (PDF) [2], are still the standard for electronic distribution of final documents for publication. Besides desktop publishing, the programming language PostScript has been used in display [3] and window systems [4] [5] as well. Nowadays it has its revival in procedural 3D modeling. The Generative Modeling Language (GML) [6] is a programming language based on PostScript. It follows the "Generative Modeling" paradigm [7], where complex data sets are represented by algorithms and parameters rather than by lists of objects. With ever increasing computing power becoming available, generative approaches [8] [9] become more important since they trade processing time for data size. At run time the compressed procedural description can be "unfolded" on demand to very quickly produce amounts of meshes, textures, etc. that are several orders of magnitude larger than the input data.

### A. PostScript in 3D

GML is very similar to Adobe's PostScript, but without any of the 2D layout operators. Instead, it provides a number of operators for creating 3D models.

PostScript and GML are interpreted, stack-based languages with strong dynamic typing, scoped memory, and garbage collection. The language syntax uses reverse Polish notation, which makes the order of operations unambiguous, but reading a program requires some practice, because one has to keep the layout of the stack in mind [10]. Most operators and functions take their arguments from the stack, and place their results onto the stack. Literals (numbers, strings, etc.) have the effect of placing a copy of themselves on the stack. Sophisticated data structures can be built on array and dictionary types, but cannot be declared to the type system. They remain arrays and dictionaries without further type information.

### B. JavaScript

PostScript programs are typically not produced by humans, but by other programs, e.g., printer drivers and devices. However, it is possible to write computer programs in PostScript just like in any other programming language.

In order to simplify the GML development and 3D design process, 3D modeling tools (Autodesk Maya, 3ds Max, etc.) can be used. Unfortunately, these tools do not preserve the generative nature. They can only export the generated result.

Encoding shape as program code clearly has the greatest flexibility, but up to now it requires coding (programming), which is usually done by humans. To accelerate the GML creation process and to increase efficiency we propose a JavaScript (JS) translator to GML. JS is a structured programming language featuring a rather intuitive syntax, which is easy to read and to understand. It also incorporates features like dynamic typing and first-class functions. The most important feature of JS is that it is already in use by many non-computer scientists, namely designers and creative coders [11]. JS and its dialects are widely used in applications and on the Internet: in Adobe Flash (called ActionScript), in interactive PDF files, in Microsoft's Active Scripting, in VRML97, etc. Consequently, a lot of documentation and tutorials to introduce the language exist [12]. In

 ISBN: 978-1-61208-112-0 

order to be used for procedural modeling, JS is missing some functionality, which we added via libraries.

### C. Overview

*Euclides* (www.cgv.tugraz.at/euclides) is a transpiling framework written in Java. *Euclides* will also translate an input JS program to Java or documents its structure in HTML. It features its own integrated development environment (IDE), from which one can transpile to the supported target languages. Our translation to GML makes the rich feature set of the Generative Modeling Language accessible to a wider range of users, because it hides much of the complexity involved in writing GML programs.

In the subsequent sections, we explain the JS to GML translator. Having parsed JS using ANTLR [13], the translation process begins with a correct (according to ECMAScript, ECMA-262, ISO/IEC 16262) Abstract Syntax Tree (AST). Then we show how data types, functions and operators are translated and explain the control flow.

## II. Data Types

In JS each variable has a particular, dynamic type. It may be `undefined`, `boolean`, `number`, `string`, `array`, `object`, or `function`. GML also has a dynamical type system. Unfortunately, both type systems are incompatible to each other. Therefore, translating JS data types to GML poses two particular problems: On the one hand, the dynamic types must be inferred at run time. On the other hand, GML's native data types lack distinct features needed by JS. GML-Strings, for example, cannot be accessed character-wise. We solved these problems by implementing JS-variables as dictionaries [6] in GML. Dictionaries are objects that map unique keys to values. These dictionaries hold needed metadata and type information as well as methods which emulate JS behavior. As we will show later, we will utilize GML's dictionaries for scoping as well.

The system translation library for GML (which every JS-translated GML program defines prior to actual program code) contains the function `sys_init_data`, which defines an anonymous data value in the sense of JS data.

```
/sys_init_data {
   dict begin
   /content dict def
   content begin
      /type  edef
      /value edef
      /length { value length } def
   end
   content
   end
} def
```

`sys_init_data` opens a new variable-scope by defining a new, anonymous dictionary and opening it. In this new scope, another newly created dictionary is defined by the name `content`. This content-dictionary receives three entries: `type`, `value` and the method `length`. Each entry value is taken from the top of GML's stack. The newly created dictionary is then pushed onto the stack and the current scope is destroyed by closing the current dictionary, leaving the anonymous dictionary on the stack. In GML notation, a JS-variable's content is defined by pushing the actual value and a pre-defined constant to identify the type of the variable (such as `Types.number`, `Types.array`, etc.) onto the stack, and calling `sys_init_data`. The translator prefixes all JS-identifiers with `usr_` (in order to ensure that all declarations of identifiers do not collide with predefined GML objects) and uses the following translations:

**Undefined**: Variables of type `undefined` result from operations that yield an undefined result or by declaring a variable without defining it. `var x;` leads to `x` being of type `undefined`. It is translated to

```
/usr_x Nulls.Types.undefined
Types.undefined sys_init_data def
```

**Boolean**: In JS, boolean values are denoted by the keywords `true` and `false`. The translation simply maps these values to equivalent numerical values in GML, which interprets them. The JS-statement `var x = true;` becomes

```
/usr_foo 1 Types.bool sys_init_data def
```

**Number**: All JS numbers (including integers) are represented as 32-bit floating point values. As GML stores numbers as 32-bit floats internally as well, we simply map them to GML's number representation. For the sake of completeness, `var x = 3.14159;` is translated to

```
/usr_x 3.14159 Types.number sys_init_data def
```

**String**: Although GML does support strings, they cannot be accessed character-wise. We cope with this limitation by defining strings as GML-arrays of numbers. Each number is the Unicode of the respective character. As GML allows to retrieve and to set array-elements based on indexes, this approach meets all conditions of JS-strings. The statement `var x = "Hello World";` becomes

```
/usr_x [ 72 101 108 108 111 32 87 111 114 108 100 ]
       Types.string sys_init_data def
```

**Array**: JS arrays allow to hold data with different types, the array's contents may be mixed. This behavior is in line with GML. The JS-example `var x = [true, false, "maybe"];` has a straightforward translation:

```
/usr_x [ 1 Types.bool sys_init_data
         0 Types.bool sys_init_data
         [109 97 121 98 101] Types.string sys_init_data ]
       Types.array sys_init_data def
```

**Object**: In JS an object consists of key-value-pairs, e.g., `var x = { x: 1.0, y: 2.0, z: 42};` This structure is mapped to nested GML-dictionaries. The value of a variable's content is a dictionary of its own. This dictionary contains the entries corresponding to JS-object's members, which are also defined as variable contents.

The example above defines a JS-object of name `x` with key-value-pairs `x` to be 1, `y` to be 2, and `z` to be 42:

 ISBN: 978-1-61208-112-0 

```
/usr_x dict begin
       /obj dict def obj begin
         /usr_x  1.0 Types.number sys_init_data def
         /usr_y  2.0 Types.number sys_init_data def
         /usr_z 42.0 Types.number sys_init_data def
       end obj
       Types.object sys_init_data end def
```

Opening an anonymous dictionary creates a new scope. In this scope, a dictionary is created and bound to the name `/obj`. It is then opened and its members are defined, just like anonymous variables would be. The object dictionary is then closed, put on the stack, and used to define an anonymous variable. The enclosing anonymous scoping dictionary is then closed and simply discarded.

JS objects may hold functions. Our translator *Euclides* handles JS object-functions like ordinary functors (next subsection) and assigns their internal name to a key-value-pair.

**Function**: JS has first-class functions. Therefore, it is possible to assign functions to variables, which can be passed as parameters to other functions, for example. In the following example, a function `function do_nothing() {}` is declared and defined. Afterwards, it is assigned to a variable `var x = do_nothing;`. If we abstract away from the translation of the function `do_nothing`, the statement `var x = do_nothing;` becomes:

```
/usr_do_nothing {
   %% ... definition of function omitted ...
} def

/usr_x /usr_do_nothing Types.function sys_init_data def
```

In JS, `x` can now be used as a functor, which acts the same ways as `do_nothing`. Because such functors can be reassigned, it is necessary to handle functor calls (`x()`) differently than ordinary function calls (`do_nothing()`). In this situation *Euclides* creates a temporary array, which contains the functor parameters and passes this array as well as the variable referencing the function name to a system function `sys_execute_var`. This system function resolves the functor and determines the referenced function, unwraps the array and performs the function call.

## III. FUNCTIONS

### A. Translation of JS Functions

In GML, functions are defined using closures, such as `/my_add { add } def`. If this function `my_add` is executed, the closure `{ add }` is put onto the stack, its brackets are removed, and the content is executed.

To execute a GML function, its parameters need to be put on the stack prior to the function call: `1.0 2.0 my_add` The resulting number `3.0` will remain on the stack. Please note, that GML functions may produce more than one result (left on the stack) at each function call. This allows to define functions with more than one result value. Following JS, called functions return only one value by convention. The number and names of function parameters are known at compile time. Only functors (referenced functions stored in variables) may change at run time and cannot be checked ahead of time.

Translated functions and parameters are named just like their JS-counterparts (except for their `usr_` prefix).

### B. Scopes

As JS uses a scoping mechanism different to GML, it has to be emulated. This is a rather difficult task, which has to take the following properties of JS scopes into account.

- JS functions may call other functions or themselves.
- Called functions may declare the same identifiers as the calling functions.
- Within functions other functions may be defined.
- Blocks might be nested inside functions, redefining symbols or declaring symbols of the same name.

The translator uses GML's dictionary mechanism to emulate JS-scopes. A dictionary on the dictionary stack can be opened and it will take all subsequent assignments to GML-identifier (variables). Since only the opened dictionary is affected, this behavior is the same as the opening and closing scopes in different scoped programming languages, such as C or Java.

Thus an assignment `/x 42 def` can be put into an isolated scope by creating a dictionary (`dict`), opening it (`begin`), performing the assignment, and closing the dictionary (`end`). The following example shows how such GML scopes can also be nested:

```
dict begin
   /x 3.141 def          %% x is 3.141
   dict begin            %%
      /x 4 def           %% x is 4.0
   end                   %% x is 3.141
end                      %% x is unknown
```

As noted before, JS supports redefinition of identifiers that were declared in a scope below the current one. Fortunately, GML exhibits just the same behavior when reading out the values of variables/keys from dictionaries of the dictionary stack. Consequently, the following example works as expected.

```
dict begin
   /x 42 def
   dict begin
      /y x 1 add def     %% y is now 43
   end
end
```

However, assignments to variables have to be handled differently in GML. The Generative Modeling Language does not distinguish between declaration and definition, any declaration must be a definition and vice versa.

The translator solves this problem. It uses a system function (which is included into all translated JS sources automatically) called `sys_def`. This function applies GML's `where` operator to the dictionary stack in order to find the uppermost dictionary, where the searched name is defined.

 ISBN: 978-1-61208-112-0 

The operator returns the reference to the dictionary, in which the name was found.

### C. Control Flow for Functions

The Generative Modeling Language and all PostScript dialects lack a dedicated jump operation in control flow. Imperative functions often require the execution context to jump to a different point in the program at any time - and to return from there as well.

Fortunately, GML provides an exception mechanism. A GML exception is propagated down GML's internal execution stack until a `catch` instruction is encountered. In this way it overrides any other control structure it encounters. We use GML's exception mechanism to jump outside a function as illustrated in the following empty function skeleton:

```
/usr_foo {
   dict begin
   /return_issued 0 def
   { dict begin
     %% ... function body omitted ...
     end }
   { /return_issued 1 def }
   catch

   return_issued not
   { Nulls.Types.undefined
     Types.undefined sys_init_data } if
   end
   sys_exception_return_handler
} def
```

In this empty skeleton, the function opens a new anonymous scope. Inside this scope `dict begin ... end` the local identifier `/return_issued` is set to `0`. Afterwards a GML try-catch-statement { *try_block* } { *catch_block* } `catch` contains the JS-function implementation. In this translation, the catch block redefines `/return_issued` to `1` to indicate that a JS `return` statement has been executed in the function body. JS functions without any `return` statement, automatically return `null` resp. in GML `Nulls.Types.undefined Types.undefined sys_init_data`. A corresponding JS-return statement, e.g., `return 42;`, is translated to

```
   42.0 Types.number sys_init_data end throw
```

In this example, the number `42.0` is put onto the stack. The actual function body's scope is closed `end`, and the `throw` operator is applied. The distinction of whether the end of the function body was reached by normal program flow or via a return statement determines, if a return value needs to be constructed (`null`) and put onto the stack.

Parameters to functions are simply put on the stack. The function body retrieves the expected number of parameters and assigns them to dictionary entries of the outer scope defined in the function translation. A complete example of a translated JS-function shows the interplay of all mechanisms. The simple JS-function

```
function foo(n) { return n; }
```

is translated to

```
/usr_foo {
   dict begin
   /usr_n edef
   /return_issued 0 def
   { dict begin
     usr_n
     end
     throw
     end }
   { /return_issued 1 def }
   catch

   return_issued not
   { Nulls.Types.undefined
     Types.undefined sys_init_data } if
   end
   sys_exception_return_handler
} def
```

A function call, for example `foo(3)`, yields the translation `3.0 Types.number sys_init_data usr_foo`. If we assign the function `foo` to a variable `foo_functor`, the calling convention in GML would change significantly.

```
/usr_foo_functor /usr_foo Types.function sys_init_data def
```

is called via

```
[ 3.0 Types.number sys_init_data ]
usr_foo_functor sys_execute_var
```

and represents the JS call `foo_functor(3.0);`

### D. Exceptions

The language JS supports throwing exceptions; e.g., `throw "Error: unable to read file.";`. Its syntax is similar to a return statement. To implement such behavior, we also use GML's exception handling mechanism. The *Euclides* translator adds a call to the predefined system function `sys_exception_return_handler` at the end of each translated function (see example above).

Throwing an exception in JS translates into a global GML variable `exception_thrown` being set to 1, closing the current dictionary and calling GML's `throw`. The `sys_exception_return_handler` will check if an actual exception is being thrown, and if so, calls `throw` again. A catch-block inside a JS program would set `exception_thrown` to 0.

## IV. Operators

The evaluation of expressions demands variables to be accessed. While GML provides operators that operate on their own set of types, they obviously cannot be used to access the translated/emulated JS-variables. For this reason, the *Euclides* translator automatically includes a set of predefined GML functions that substitute operators defined in JS.

### A. Value Access

Performing the opposite operation to `sys_init_data`, `sys_get_value` will retrieve the data saved in a JS-variable resp. its GML-dictionary. For example, to retrieve `v.value` the function `sys_get_value` is applied to `v`.

```
/sys_get_value { begin value end } def
```

 ISBN: 978-1-61208-112-0 

### B. Element Access

The system function `sys_get` implements string, array and object access. Applied to a string / an array `Arr` and index `k`, it will return the element `Arr[k]`. If its parameters are an object `Obj` and an attribute `name`, the function `sys_get` executes `Obj.name`. This may result in a value, which is put on the stack or in a function, which is called. Conforming to JS, it returns JS `undefined` for any requested elements that do not exist.

```
/sys_get {
   dict begin
   /idx exch def /var exch def

   var.type Types.string eq {
      %% ... handling strings ...
   } if

   var.type Types.array  eq {
      %% ... handling arrays  ...
   } if

   var.type Types.object  eq {
      var sys_get_value idx known 0 eq {
         %% return null, if element doesn't exist
         Nulls.Types.undefined
         Types.undefined sys_init_data
      } if
      var sys_get_value idx known 0 ne {
         %% access element
         var sys_get_value idx get
      } if
   } if
   end
} def
```

Analogous to `sys_get`, `sys_put` inserts data into strings and arrays, or defines members of objects. If `sys_put` encounters an index $k$ that is out of an array's range, the array is resized and filled with JS `undefined`s.

### C. Functors

The already mentioned routine `sys_execute_var` inspects a given variable. If it is a function, it will retrieve the array supplied to hold all parameters and execute the function. The dynamic binding of functions to variables requires to consider two situations at run time: The functor receives the correct amount of parameters for its function, or the number of parameters does not correspond to the referenced function. In the later case, the function is not called and `null` is returned instead.

At compile time, a function is defined to expect a concrete number of parameters. This information is kept to perform parameter checks at run time. In this way, the correct number of parameters for all functors can be determined any time.

### D. JS built-in Operators

To illustrate the translation of relational, arithmetical or bit-shift operators defined by JS, we discuss the equal operator `==`. It is (like all such operators) mapped to a corresponding routine `sys_eq`. Depending of the operands' types it delegates the comparison to subroutines such as `bool_eq`, `string_eq` or `array_eq` that perform the actual comparison. If the types and the values do match, `sys_eq` directly returns the JS-value `true`. If types do not match, the variable is converted to the type of the respective operand, as specified by JS, and then compared.

## V. Control Flow

### A. Conditional Statement

The JS if-then-else statement corresponds one-to-one to the same GML statement. Consequently, the conditional expression is translated straightforwardly. Using the expression mapping introduced in the previous section (e.g. `sys_eq` implements the equality operator), the JS statement `if(a == b) { c = a; } else { c = b; }` is translated info:

```
%% if (a==b)
usr_a usr_b sys_eq sys_get_value
{ %% then:
   dict begin {
      dict begin
         /usr_c usr_a sys_def
      end
   } exec end
}
{ %% else:
   dict begin {
      dict begin
         /usr_c usr_b sys_def
      end
   } exec end
} ifelse
```

The exec-statements (and their closures) stem from the fact that both sub-statements, the then-part and the else-part, are statement blocks `{ ... }`. These blocks are executed within their own, new scopes.

### B. Loops

GML supports different types of looping control structures, which have similar names to JS-loops (e.g., both languages have a for-loop). However, the GML counterparts have different semantics (e.g., GML's for-loop has a fixed, finite number of iterations, which is known before execution of the loop body, whereas JS-loops evaluate the stop condition during execution, which may result in endless loops). The *Euclides* translator uses the GML `loop` mechanism, which is an infinite loop that can be quit using the `exit` operator.

An important problem is that control structures such as for, while and do-while are not only controlled by the loop's stop condition, but also by JS statements such as `continue` and `break` within the loop body (besides `return` and `throw` as mentioned before). The statement `break` immediately stops execution of the loop and leaves it, whereas `continue` terminates the execution of the current loop iteration and continues with the next iteration of the loop. Therefore, we translate an empty while loop `while(false) { ... }` to

```
{  /continue_called 0 def
   {  0 Types.bool sys_init_data
      sys_get_value not { exit } if
      {  dict begin
         %% ... loop body omitted ...
         end
```

 ISBN: 978-1-61208-112-0 

```
      } exec
   } loop
   continue_called not { exit } if
} loop
```

GML's `exit` keyword terminates the current loop. This behavior is leveraged by the *Euclides* translator to implement `break` and `continue`. The translation uses two nested loops that will run infinitely.

Prior to the begin of the inner loop `/continue_called` is set to `0`. At the top of the inner loop, the loop condition is tested. If the condition evaluates to `false`, the inner loop is exited using GML's `exit`. Otherwise a new scope is created and the loop-statement executed within that scope.

During loop iterations, there are three scenarios under which a loop can terminate:

1) If the loop condition is met: When the condition evaluates to `false`, the inner loop is exited. Since `continue_called` is not set to `true`, the outer loop will terminate as well.
2) If the loop body encounters JS `break` (resp. GML `exit`): Again, the inner loop is left. `continue_called` will not be set to `true`, hence the outer loop will also terminate.
3) If the function returns: GML's exception throwing mechanism will unwind the stack until the catch-handler at the end of the function is encountered.

If the loop body encounters a JS-`continue` statement, `continue_called` will be set to `true` and the GML `exit` command will immediately stop the inner loop. Since `continue_called` is set, execution does not leave the outer loop, however. As a consequence, `continue_called` becomes `0` again, and execution re-enters the inner infinite loop.

The do-while-statement is translated very similar to the while-statement. The only semantic differences in JS are that execution will enter the loop regardless of the loop-condition and that the loop-condition is tested after loop body execution. *Euclides* translates an empty do-while-statement `do { ... } while (false)` as follows:

```
{  /continue_called 0 def
   {  {  dict begin
         %% ... loop body omitted ...
         end
      } exec
      0 Types.bool sys_init_data
      sys_get_value not { exit } if
   } loop
   continue_called not { exit } if
   0 Types.bool sys_init_data
   pop
} loop
```

Due to a semantic difference of JS `continue` in do-while-loops, this statement needs to be handled differently. If `continue` is encountered, the loop condition must still execute before the loop body is re-entered, because side effects inside the loop condition may occur (such as incrementing a counter). *Euclides* handles this problem by executing the condition expression a second time in the outer loop. Since expressions always return values, any value resulting from the loop-expression has to be popped off the stack.

Although GML has a `for` operator, it is semantically incompatible with JS's one. Its increment is a constant number, and so is the limit. In JS, both increment and limit must be evaluated at each loop body execution. Therefore, we translate `for` just like the previous constructs by two nested loops with the increment condition repeated in outer loop (due to `continue` semantics). Finally, *Euclides* translates the JS statement `for (var i=0; i < 1; i++) { }` to GML via

```
dict begin
%% initialization (i=0)
/usr_i 0.0 Types.number sys_init_data def
{  /continue_called 0 def
   {  %% condition (i<1)
      usr_i 1.0 Types.number sys_init_data sys_lt
      sys_get_value not { exit } if
      {  dict begin
         %% ... loop body ...
         end
      } exec
      %% increment (i++)
      usr_i
         usr_i 1 Types.number sys_init_data sys_add
      /usr_i sys_edef
      pop
   } loop
   continue_called not { exit } if
   %% increment again (i++)
   usr_i
      usr_i 1 Types.number sys_init_data sys_add
   /usr_i sys_edef
   pop
} loop
end
```

The JS for-in statement `for(var x in array) statement;` is semantically equivalent to:

```
for (var i = 0; i < array.length; i++) {
   var x=array[i]; statement;
}
```

This construction loops over the elements of an array provides the loop body with a variable holding the current element.

### C. Selection Control Statement

The translation of the JS `switch` statement poses several difficulties:

- If a case condition is met, execution can "fall through" till the next `break` is encountered.
- If a `break` is encountered, the currently executed `switch` statement must be terminated.
- Of course, `switch` statements may be nested.

To develop a semantically consistent solution, we did not want to alter the translation of JS-`break` inside switch statements (compared to loops). We solve the problem of breaking outside the `switch` statement by implementing it as a loop that is run exactly once. In GML it reads like `1 { loop_instructions } repeat`. This way our translation of `break` shows semantically correct behavior, it terminates the loop. Consider the following JS-program:

 ISBN: 978-1-61208-112-0 

```
var x = 0, y = 0;

function bar() { return 3; }

function foo(i) {
   switch(i) {
      case 0:
      case 1:
      case 2: x = 1;
      case 4: x = 3;
      case bar(): x = 2; break;
      default: y = 5;
} }
```

The function `foo` will be translated to:

```
/usr_foo
{  dict begin
   /usr_i edef
   /return_issued 0 def
   { dict begin
      /switch_cnd_met1 0 def
      1 { usr_i 0.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
          } if

          usr_i 1.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
          } if

          usr_i 2.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 1;
             /usr_x 1.0 Types.number
             sys_init_data sys_def
          } if

          usr_i 4.0 Types.number sys_init_data sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 3;
             /usr_x 3.0 Types.number
             sys_init_data sys_def
          } if

          usr_i usr_bar sys_eq
          sys_getvalue switch_cnd_met1 1 eq or {
             /switch_cnd_met1 1 def
             %% x = 2;
             /usr_x 2.0 Types.number
             sys_init_data sys_def
             exit
          } if
       %% y = 5;
       /usr_y 5.0 Types.number
       sys_init_data sys_def
   } repeat
   currentdict /switch_cnd_met1 undef end
   }
   { /return_issued 1 def } catch

   return_issued not {
      Nulls.Types.undefined
      Types.undefined sys_init_data
   } if
   end
   sys_exception_return_handler
} def
```

This example shows that we introduce an internal variable `/switch_cnd_metX` for traversing the case statements. As soon as a case statement condition is met, `/switch_cnd_metX` is set to `true`, leading execution into every encountered case statement.

The *Euclides* translator takes into account that switch statements may be nested. As it traverses the AST, it keeps book of all internal variable to ensure a unique name (`switch_cnd_met1`, `switch_cnd_met2`, ..., `switch_cnd_metN`).

The example translation shows that for `foo(3)` the cases 0, 1, 2, 4 and 3 (= `bar()`) will only execute case 3, where the `1 { } repeat` statement will be broken out of with the GML `exit` operator. The default block will be executed in any case if execution is still inside the `repeat` statement, no further state is checked for `default`.

## VI. EXAMPLE

To demonstrate the interplay of all translational building blocks, this section shows a non-recursive, subtraction-based version of the Euclidean algorithm to calculate the greatest common denominator and its translation to GML. It can be shown by induction that two successive Fibonacci numbers are the computational worst-case of the Euclidean algorithm. We use them as input data.

```
function fibonacci(index) {
   switch (index) {
      case  0:
      case  1: return 1;
      default: return  fibonacci(index-2)
                     + fibonacci(index-1);
   }
}

function gcd(a,b) {
   if (a == 0) return b;
   while (b != 0)
      if (a > b) a = a - b;
      else       b = b - a;
   return a;
}

var x = gcd(fibonacci(5), fibonacci(6));
```

The corresponding GML code is:

```
/usr_fibonacci {
   dict begin
   /usr_index edef
   /return_issued 0 def
   {  dict begin
      /switch_cnd_met1 0 def
      1 {usr_index 0.0 Types.number sys_init_data
         sys_eq sys_getvalue switch_cnd_met1 1 eq or {
            /switch_cnd_met1 1 def
         } if

         usr_index 1.0 Types.number sys_init_data
         sys_eq sys_getvalue switch_cnd_met1 1 eq or {
            /switch_cnd_met1 1 def
            1.0 Types.number sys_init_data
            end throw
         } if

         usr_index 2.0 Types.number sys_init_data
         sys_sub usr_fibonacci
         usr_index 1.0 Types.number sys_init_data
         sys_sub usr_fibonacci
         sys_add
         end throw
      } repeat
      currentdict /switch_cnd_met1 undef end
   }
   { /return_issued 1 def } catch
   return_issued not {
```

```
      Nulls.Types.undefined
      Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def

/usr_gcd {
   dict begin
   /usr_a edef
   /usr_b edef
   /return_issued 0 def
   {  dict begin
      usr_a 0.0 Types.number sys_init_data
      sys_eq sys_getvalue
      { usr_b end throw }
      {}
      ifelse

      { /continue_called 0 def
        {  usr_b 0.0 Types.number sys_init_data
           sys_ne sys_getvalue not { exit } if

           usr_a usr_b sys_gt sys_getvalue
           {  /usr_a usr_a usr_b sys_sub sys_def }
           {  /usr_b usr_b usr_a sys_sub sys_def }
           ifelse exec
        } loop
        continue_called not { exit } if
   } loop
   usr_a end throw
   end
   }
   { /return_issued 1 def } catch
   return_issued not {
      Nulls.Types.undefined
      Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def

/usr_x
  6.0 Types.number sys_init_data usr_fibonacci
  5.0 Types.number sys_init_data usr_fibonacci
  usr_gcd
def
```

## VII. Conclusion

In this article, we presented a JS to PostScript translator. While this translation is a simple infix-to-postfix notation rewrite for mathematical expressions (`1+2` becomes basically `1 2 add`), the correct translation of control flow structures is a non-trivial task, due to the fact that there is no concept of `goto` in the PostScript language and its dialects.

The main contribution of this work is the complete translation of JS into a PostScript dialect including *all* control flow statements. To the best of our knowledge, this is the first *complete* translator. Other projects (PdB by Arthur van Hoff, pas2ps by Dulith Herath and Dirk Jagdmann) do not support, e.g., `return` statements.

As *Euclides* offers a new access to GML, all GML users will benefit from its results. The possibility to use GML via a JS-to-GML translator reduces the inhibition threshold significantly. Everyone, who knows any imperative, procedural language (Pascal, Fortran, C, C++, Java, etc.) is familiar with the language concepts in JS and can use *Euclides*. Advanced GML users, who already know how to program in PostScript style, can use *Euclides* to translate algorithms, which are often presented in a imperative, procedural (pseudo-code) style [14].

## Acknowledgment

We would like to thank Richard Bubel for his valuable support on ANTLR and the JS grammar. In addition, the authors gratefully acknowledge the generous support from the European Commission for the integrated project 3D-COFORM (www.3Dcoform.eu) under grant number FP7 ICT 231809, from the Austrian Research Promotion Agency (FFG) for the research project METADESIGNER, grant number 820925/18236, as well as from the German Research Foundation (DFG) for the research project PROBADO under grant INST 9055/1-1 (www.probado.de).

## References

[1] Adobe Systems, Inc., *PostScript Language Reference Manual (first ed.)*. Addison-Wesley, 1985.

[2] "Document management – Portable Document Format," 2008.

[3] Adobe Systems, Inc., *Display PostScript System*. Adobe Systems Incorporated, 1993.

[4] J. Gosling, "SunDew – A Distributed and Extensible Window System," *Methodology of Window Management (Eurographics Seminars); Proceedings of an Alvey Workshop at Cosener's House, Abingdon, UK*, vol. 5, pp. 1–12, 1986.

[5] C. Geschke, S. McGregor, J. Gosling, L. Hourvitz, and M. Callow, "Screen postscript," *International Conference on Computer Graphics and Interactive Techniques archive; ACM SIGGRAPH 88 panel proceedings*, vol. 22, pp. 1–43, 1988.

[6] S. Havemann, "Generative Mesh Modeling," *PhD-Thesis, Technische Universität Braunschweig, Germany*, vol. 1, pp. 1–303, 2005.

[7] J. M. Snyder and J. T. Kajiya, "Generative modeling: a symbolic system for geometric modeling," *Proceedings of 1992 ACM Siggraph*, vol. 1, pp. 369–378, 1992.

[8] P. Müller, P. Wonka, S. Haegler, U. Andreas, and L. Van Gool, "Procedural Modeling of Buildings," *Proceedings of 2006 ACM Siggraph*, vol. 25, no. 3, pp. 614–623, 2006.

[9] S. Havemann and D. W. Fellner, "Generative Parametric Design of Gothic Window Tracery," *Proceedings of the 5th International Symposium on Virtual Reality, Archeology, and Cultural Heritage*, vol. 1, pp. 193–201, 2004.

[10] G. C. Reid, *Thinking in Postscript*. Addison-Wesley, 1990.

[11] C. Reas, B. Fry, and J. Maeda, *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2007.

[12] E. A. Vander Veer, *JavaScript for Dummies*. For Dummies, 2004.

[13] T. Parr, *The Definite ANTLR Reference – Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[14] T. H. Cormen, C. Stein, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. B&T, 2001.

 ISBN: 978-1-61208-112-0 