Implementing a WebIDL compiler for Jerryscript*

9/21/2017

Tim Harvey

** Unapologetically cribbed from an earlier talk... ☺*

Introduction

I :

- Graduated from Rice University (1988, 1998, 2003)
  - Computer Science
    - Compilers
      - Data-flow analysis, SSA, (graph-coloring) register allocation
- Worked at Rice from 1991 – 2012 for Keith Cooper (21 years)
- Have been at TI 5+ years
  - Led the optimizer group
  - Currently working on the “scripting project”

What _is_ the “scripting project”?

In a nutshell:

We would like users to be able to program our chips with high-level scripts

1. More user friendly
2. Faster prototyping

We would like minimal involvement from our library developers/maintainers

This is fundamentally a compiler problem.

Myriad Options in languages/environments

- Lua
- Elua
- Pawn
- Pymite / Micropython
- B#
- Rust
- MY_BASIC/TinyBasic
- Tcl * / Ficl
- Tiny-JS/Mu-JS/Duktape/V8
- Picobit
- EmbedVM / NanoVM
- Wren
- SX-Forth /AmForth /ZForth
- Bitlash
- AngelScript
- PicoC
- Interactive C
- Armpit Scheme
- Jx9
- Script
- Embedded Ch
- Rappit
- Mruby
- Lily
- V8 (EcmaScript)

Requirements

1. Must run on small-memory architectures
2. Should be easy
3. Should be popular
4. Should be powerful/expressive
5. Open source

...So what do we do?...

Questions:

1. Who is our target market?
2. What is our monetization strategy?

What we chose

1. Javascript
    1. Ubiquitous
    2. IOT
    3. Linaro/Zephyr
2. Jerryscript
    1. Supported by Samsung/Intel/et al.
    2. Actively developed
    3. Small(!)
    4. C API

Calculator Example

In Javascript:

```
var Calculator = new Object();
Calculator.Prototype.add = function (x, y) {return x+y;};
Calculator.Prototype.subtract = function (x, y) {return x-y;};
var new_calc = new Calculator;
console.log(new_calc.add(4, 5));
```

Calculator Example

In Javascript:

```
var Calculator = new Object();
Calculator.Prototype.add = function (x, y) {return x+y;};
Calculator.Prototype.subtract = function (x, y) {return x-y;};
var new_calc = new Calculator;
console.log(new_calc.add(4, 5));
```

Using the Jerryscript API:

```
jerry_value_t new_object = jerry_create_object();
```

Calculator Example

In Javascript:

```
var Calculator = new Object();
Calculator.Prototype.add = function (x, y) {return x+y;};
Calculator.Prototype.subtract = function (x, y) {return x-y;};
var new_calc = new Calculator;
console.log(new_calc.add(4, 5));
```

Using the Jerryscript API:

```
jerry_value_t function = jerry_create_external_function(add_handler);
jerry_set_property(new_object, "add", function);
```

Calculator Example

In Javascript:

```
var Calculator = new Object();
calculator.Prototype.add = function (x, y) {return x+y;};
calculator.Prototype.subtract = function (x, y) {return x-y;};
var new_calc = new Calculator;
console.log(new_calc.add(4, 5));
```

Using the Jerryscript API:

```
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
        ...

} /* add_handler */
```

Calculator Example – add()

In C:

```
/* from the Javascript: z = calculator.add(x, y)*/
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x + y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* add_handler */
```

Calculator Example – add()

In C:

```
/* from the Javascript: z = calculator.add(x, y)*/
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x + y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* add_handler */
```

Calculator Example – add()

In C:

```
/* from the Javascript: z = calculator.add(x, y)*/
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x + y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* add_handler */
```

Calculator Example – add()

In C:

```
/* from the Javascript: z = calculator.add(x, y)*/
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x + y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* add_handler */
```

Calculator Example – add()

In C:

```
/* from the Javascript: z = calculator.add(x, y)*/
jerry_value_t add_handler(jerry_value_t object, jerry_value_t argv, int argc)
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x + y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* add_handler */
```

Calculator Example – subtract()

In C:

```
/* from the Javascript: z = calculator.subtract(x, y)*/
jerry_value_t subtract_handler(jerry_value_t object, jerry_value_t argv, int a
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x - y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* subtract_handler */
```

Boilerplate for the API:

```
/* from the Javascript: z = calculator.subtract(x, y)*/
jerry_value_t subtract_handler(jerry_value_t object, jerry_value_t argv, int a
{
    int x = jerry_get_number_value(argv[0]);
    int y = jerry_get_number_value(argv[1]);

    int z = x - y;

    jerry_value_t  return_value = jerry_create_number(z);
    return return_value;

} /* subtract_handler */
```

...This suggests an automated solution... But what?

TI's focus

Problem:

We have libraries written in C for inclusion in C programs.

Rewriting each library in one or more new languages is cost prohibitive.

Solution:

Use interpreters' C APIs to create *language extensions*

What we've got so far:

1. Interpreter
2. C APIs
3. C libraries
4. Boilerplate

...but boilerplate is still expensive, error prone, *etc*.

WebIDL

Web Interface Definition Language

- Defines APIs
- Has its own (active) standards body
- C-ish syntax
- Own type system (different from C and Javascript)
- Has two main constructs:
    1. definitions (C structs)
    2. interfaces (C++ classes)
        1. operations (C++ methods)
        2. attributes (C++ member (instance) variables)

WebIDL, calculator example

```
interface Calculator {
        long add(long x, long y);
        long subtract(long x, long y);
};
```

Finding the tool

Problem: I need a tool that will parse WebIDL and output Jerryscript boilerplate

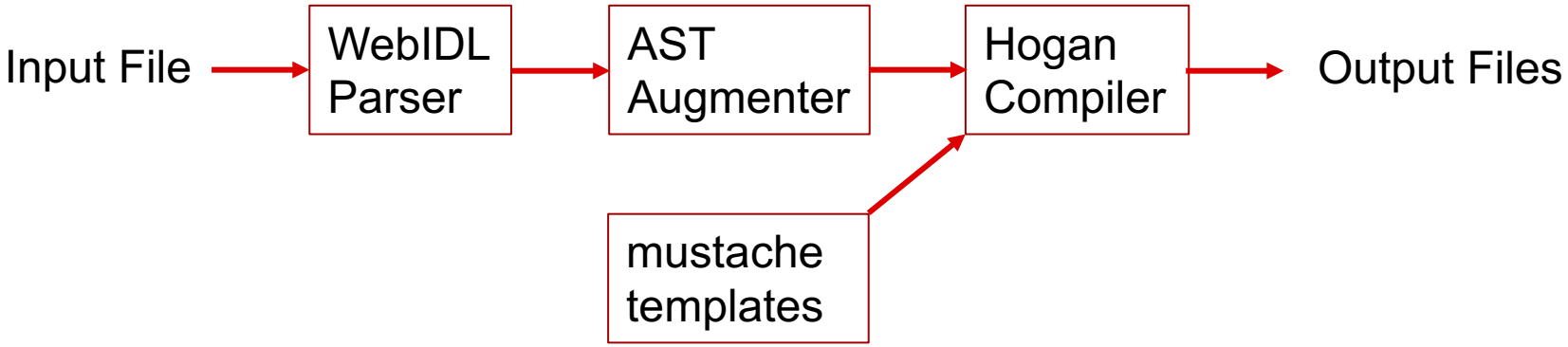
Problem: this tool does not exist – I can:

1. Write my own
2. Find a piece of open-source software
   - But what to look for?
   - How to cull? (tons of people (Chrome, FoxFire, *etc*.) play with WebIDL)

In the end, I chose a project done by a Masters student in England named Mohamed Eltuhamy:

1. It is abandoned open-source software; it was only half(?) completed
2. It does a superset of what I need (although it's close)

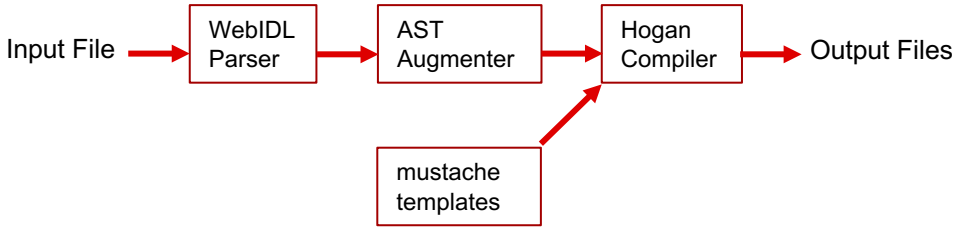
The “generator”



Notes:

1. The WebIDL parser is the official parser of the W3C standard
2. The Hogan compiler compiles mustache code/templates

The “generator”



Notes:

1. Parser/Augmenter/Hogan written in Javascript

2. The Hogan compiler compiles mustache code
   - ”logic-less” templates

3. Simple syntax:
   generate.js –package=<package_name> <idl_filename.idl>

4. Produces:
   1. <package>_Types.[ch]
   2. One <interface_name>.c and <interface_name>_stubs.c for each interface

Calculator example

```
>>> generate.js --package=Calculator Calculator.idl

Creating directory... (/temp/Calculator)
Creating C Stubs File: >/temp/Calculator/Calculator_stubs.c<
Creating C File... (/temp/Calculator/Calculator.c)
Creating header file... (/temp/Calculator/Calculator_Types.h)
```

Calculator example – Calculator_Types.h

```
typedef struct {    /* USER CODE GOES HERE */} Native_Object_Calculator;

Native_Object_Calculator *Native_Object_Calculator_create(void);

typedef struct {Native_Object_Calculator *native_object;} Calculator;


void Native_Object_set(void *thing, jerry_value_t object,
                       jerry_object_native_info_t *checksum);

void *Native_Object_get(jerry_value_t object,
                        jerry_object_native_info_t *checksum,
                        jerry_error_t *error_value);


Calculator jerry_get_Calculator_value(jerry_value_t value);

jerry_value_t jerry_create_Calculator(Calculator x);


void load_all_Calculator_interfaces(void);
```

Calculator example – Calculator.c

```
void load_Calculator_interface(void)
{
    /* first, make sure that this prototype doesn't already exist */
    jerry_value_t existing_prototype = get_prototype((char *)"Calculator");
    if (!jerry_value_has_error_flag(existing_prototype))
    {
        jerry_release_value(existing_prototype);
        return;
    }
    jerry_value_t global_object = jerry_get_global_object();
    /* add all of the interface prototypes */
    jerry_value_t Calculator_prototype_object = jerry_create_object();
    add_field_to_object(Calculator_prototype_object,
                        "add", &add_handler);
    add_field_to_object(Calculator_prototype_object,
                        "subtract", &subtract_handler);
    register_prototype((char *)"Calculator", Calculator_prototype_object);
    jerry_release_value(Calculator_prototype_object);
    add_field_to_object(global_object, "Calculator",
                        &create_Calculator_interface_handler);
    jerry_release_value(global_object);
} /* load_Calculator_interface */
```

Calculator example – Calculator.c

```
jerry_value_t
create_Calculator_interface_handler(const jerry_value_t func_value,
                    const jerry_value_t this_val,
                    const jerry_value_t *args_p,
                    const jerry_length_t args_cnt)
{
    jerry_value_t new_Calculator = jerry_create_object();
    jerry_value_t prototype = get_prototype((char *)"Calculator");
    jerry_release_value(jerry_set_prototype(new_Calculator,
                                            prototype));
    jerry_release_value(prototype);

    /* setup the Native_Object for the new object */
    Native_Object_Calculator *native_object=Native_Object_Calculator_create();
    Native_Object_set(native_object, new_Calculator, &Calculator_checksum);

    return new_Calculator;
} /* create_Calculator_interface_handler */
```

Calculator example – Calculator.c

```
static jerry_value_t add_handler(const jerry_value_t func_value
                  const jerry_value_t this_val,
                  const jerry_value_t *args_p,
                  const jerry_length_t args_cnt)
{
    /* demarshal the arguments */
    int32_t x = jerry_get_int32_t_value(args_p[0]);
    int32_t y = jerry_get_int32_t_value(args_p[1]);

extern int32_t Calculator_add_body(int32_t, int32_t, jerry_value_t);
    int32_t return_value = Calculator_add_body(x, y, this_val);

    return jerry_create_number(return_value);

} /* add_handler */
```

Calculator example – Calculator_stubs.c

```
static void Native_Object_Calculator_deallocator(void *native_object)
{
        /* USER CODE GOES HERE */
} /* Native_Object_Calculator_deallocator */

Native_Object_Calculator *Native_Object_Calculator_create(void)
{
    Native_Object_Calculator *new_object =
                                    malloc(sizeof(Native_Object_Calculator));
        /* USER CODE GOES HERE */
     return new_object;
} /* Native_Object_Calculator_create */

jerry_object_native_info_t Calculator_checksum =
                                        {Native_Object_Calculator_deallocator};
```

Calculator example – Calculator_stubs.c

```
#include "Calculator_Types.h"

/* Calculator */
int32_t Calculator_add_body(int32_t x, int32_t y, jerry_value_t this_val)
{
    /* EXAMINE THE VALUE OF "ERROR_CHECK" IF THERE COULD BE AN ERROR
       WITH AN OBJECT'S Native_Object */
    Native_Object_Calculator *native_object =
                          Native_Object_get(this_val, &Calculator_checksum,
                                            &error_check);

    /* USER CODE GOES HERE */

}; /* Calculator_add_body */
```

Calculator example – Calculator_stubs.c

```
#include "Calculator_Types.h"

/* Calculator */
int32_t Calculator_add_body(int32_t x, int32_t y, jerry_value_t this_val)
{
    /* EXAMINE THE VALUE OF "ERROR_CHECK" IF THERE COULD BE AN ERROR
       WITH AN OBJECT'S Native_Object */
    Native_Object_Calculator *native_object =
                          Native_Object_get(this_val, &Calculator_checksum,
                                            &error_check);

    return x + y;

}; /* Calculator_add_body */
```

Calculator example – how to use

1. In Calculator_Types.h is a macro called load_all_Calculator_interfaces
2. Insert that call into main.c:
3. Compile together main.c and all of the files created by the generator

```
int main()
{
  /* Initialize engine */
    jerry_init (JERRY_INIT_EMPTY);

    /* set up language extensions */
    load_all_Calculator_interfaces;

    /* test the code */
...
```

Callbacks

```
callback PrintCallback1 = void (complex x);
interface Complex_Calculator {
...
    attribute PrintCallback1 print_it;
    void      add_and_print(complex x, complex y, PrintCallback1 print_it);
...
};
```

```
void Complex_Calculator_add_and_print_body(complex x, complex y,
                  PrintCallback1_calling_context print_it_context,
                  jerry_value_t this_val)
{
#define print_it(...) (run_PrintCallback1_function(print_it_context,
                                                   __VA_ARGS__))
   ...
   print_it((complex){x.real+y.real, x.imag+y.imag});

}; /* Complex_Calculator_add_and_print_body */
```

Callbacks

```
callback PrintCallback1 = void (complex x);
interface Complex_Calculator {
...
    attribute PrintCallback1 print_it;
    void      add_and_print(complex x, complex y, PrintCallback1 print_it);
...
};
```

```
void Complex_Calculator_add_and_print_body(complex x, complex y,
                  PrintCallback1_calling_context print_it_context,
                  jerry_value_t this_val)
{
#define print_it(...) (run_PrintCallback1_function(print_it_context,
                                                   __VA_ARGS__))
   ...
   print_it((complex){x.real+y.real, x.imag+y.imag});

}; /* Complex_Calculator_add_and_print_body */
```

Callbacks

```
callback PrintCallback1 = void (complex x);
interface Complex_Calculator {
...
    attribute PrintCallback1 print_it;
    void      add_and_print(complex x, complex y, PrintCallback1 print_it);
...
};
```

```
voidrun_PrintCallback1_function(const
                  PrintCallback1_calling_context PrintCallback1_context,
                  complex x)
{
    jerry_value_t x_value = jerry_create_complex(x);

    jerry_value_t argv[] = { x_value };

    jerry_value_t jerry_return_value =
         jerry_call_function(PrintCallback1_context.function_value,
                             PrintCallback1_context.this_value, argv, 1);

} /* run_PrintCallback1_function */
```

Callbacks

```
callback PrintCallback1 = void (complex x);
interface Complex_Calculator {
...
    attribute PrintCallback1 print_it;
    void      add_and_print(complex x, complex y, PrintCallback1 print_it);
...
};
```

```
typedef struct {
    jerry_value_t function_value; /* Jerryscript's function pointer */
   jerry_value_t this_value;     /* i.e., "this" pointer */
}

typedef callback_context PrintCallback1_calling_context;
```

Design (questions/problems/goals/*etc*.)

In no particular order:

1. Not all types are handled (promises, “any”)
2. Not all of WebIDL is supported
3. Not all of Javascript is supported
4. Model is compile-once (*i.e.*, no dynamic loading)
5. We currently compile to C; C++ would be easier
6. Mustache code is unreadable
   - Code maintenance is difficult
   - Unit-testing is difficult
7. Schizophrenic handling of error checking, et al.

Uncertainties/Observations

In no particular order:

- How is multi-threading handled? i.e., what memory is shared, what is private?
- How (how much) does the user/maintainer interact with Jerryscript?

Things I find congenial in Jerryscript

In no particular order:

- Automatic parameter checking
- Evolution is in the right directions
- The code is solid and easy to work with
- The API is largely unchanging (or only grows)

Things I find irksome in Jerryscript

In no particular order:

- Setting fields (methods, values) is a multi-step process
- Likewise, I don't always want a jerry_value_t back
  (procedure/function difference)
- Documentation is unhelpful

Thank You!