Overview of JerryScript Internals

Zoltan Herczeg
zherczeg.u-szeged@partner.samsung.com

Samsung Research Hub @ University of Szeged

JerryScript Developer Meeting 2016
Staines, UK, April 26, 2016

Overview

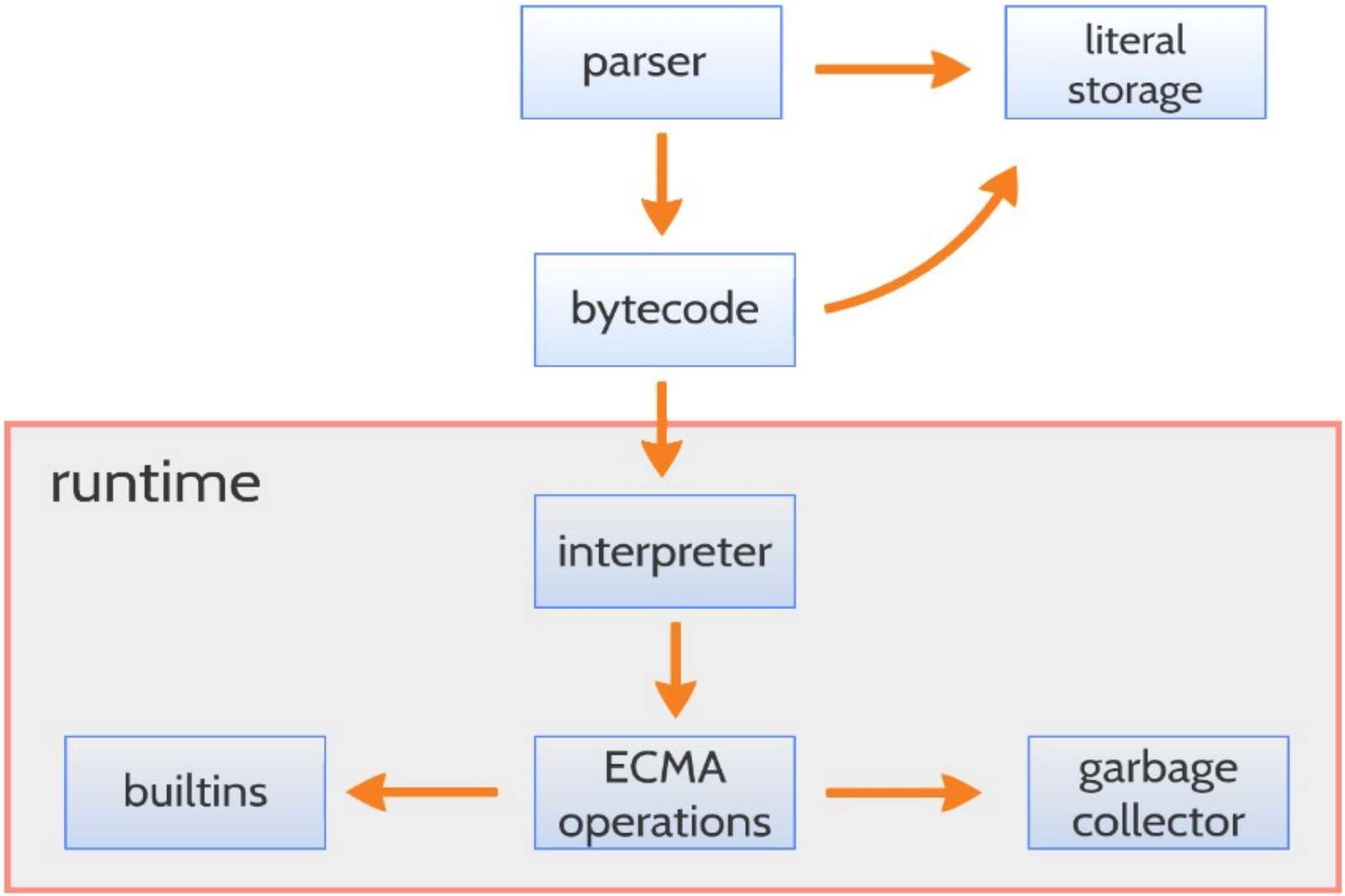
- Introduction
- Parser and interpreter
- ECMAScript data representation
- Named property resolution
- Summary

Introduction

Introduction

- JerryScript is a lightweight ECMAScript 5.1 engine, which is optimized for low-end systems
  - Embedded systems with 32 bit CPU and 64K or less RAM
- Small binary size
  - Only 173Kbyte on ARM
- Development started by Samsung
- Open source
  - https://github.com/Samsung/jerryscript

High-Level Design Overview

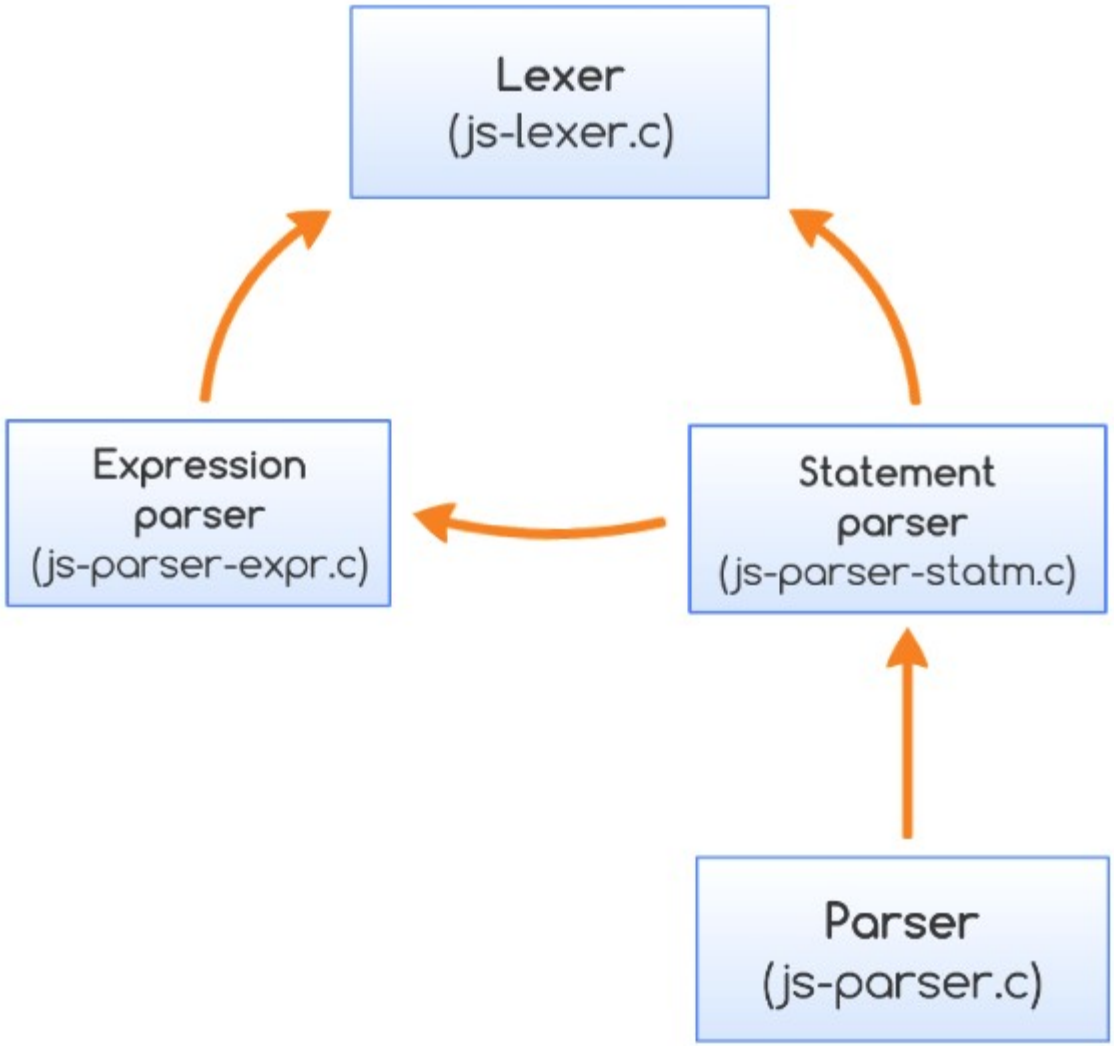


Parser and Interpreter

Parser Overview

- Optimized for low memory consumption
  - E.g. only 41Kbyte memory is required to parse the 95Kbyte concatenated IoT.js source code
    - 12.5Kbyte byte code, 10Kbyte literal references, 12.2Kbyte literal storage data, 7Kbyte for parser temporaries
- Generates byte-code directly
  - No intermediate representation (e.g. AST)
- Recursive descent parser
  - The parser uses a byte array for the parser stack instead of recursively called functions

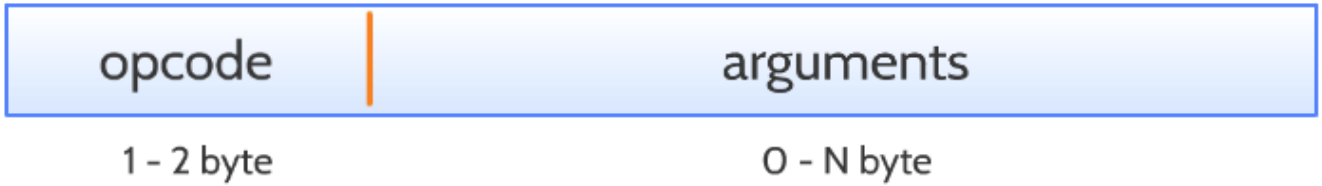
Key Modules of the Parser



Parser Overview (2)

- Has a speculative scanner
  - Checking whether for statements are for-in statements
  - Finding case / default labels of a switch statement
  - Checking tokens are not enough because a slash (/) character can be a division operator or a regular expression start
- Generated code format of the main or eval program and all functions:

Compact Byte Code (CBC)



- CBC is a unique variable length byte code with lightweight data compression
- Currently 306 opcodes are defined
  - Majority of the opcodes are variants of the same operation
- E.g. “this.name” is a frequent expression in JavaScript, so an opcode is defined to resolve this expression
  - Usually this operation is constructed from multiple opcodes: op_load_this, op_load_name, op_resolve
  - Other examples: “a.b(c,d)” or “i++”

Compact Byte Code (CBC) (2)

- Analysing JS source files can reveal these frequent constructs. Defining separate opcodes for them reduces the byte-code footprint
- Parser opcode morphing
  - The last opcode is not pushed into the byte code stream
  - Instead, it can be changed based on the next token
- Example: parsing a “this.a = …” statement
  - The lexer returns with “this” token first: CBC_PUSH_THIS
  - Followed by “.a”: CBC_PUSH_PROP_THIS_LITERAL
  - Next is equal sign: CBC_ASSIGN_PROP_THIS_LITERAL

Compact Byte Code Interpreter

- The interpreter is a combination of register and stack machines
  - Stack is used for computing temporary values
  - Registers are used for storing local variables
- Byte-code decompression
  - Byte codes are decoded into a maximum of three atomic opcodes and these opcodes are executed
- The main loop is non-recursive to reduce stack usage
  - Function calls are handled by another subsystem

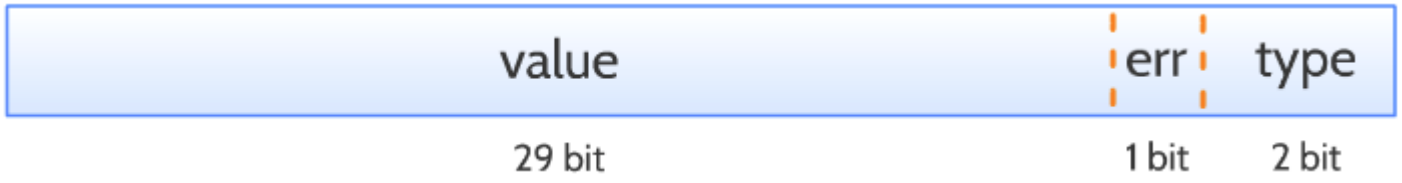
ECMAScript Data Representation

Compressed Pointers

- Compressed pointers are 16 bit values, which represent 8 byte aligned addresses in the Jerry heap
  - On 32 bit systems these pointers consume half of the space required by normal pointers
- Jerry heap is a linear memory space with a maximum size of 512Kbyte (equals to UINT16_MAX * 8)
  - UINT16_MAX is 65535

ECMA Value Representation

- JavaScript is a dynamically typed language
  - All values must provide their types as well
- ECMAScript Values in Jerry are 32 bit values
  - They can be simple values (true, null, undefined, …), or pointers to numbers, strings, or objects
- On 32 bit systems, 29 bit is enough to directly store any 8 byte aligned 32 bit pointers
  - Otherwise a compressed pointer is stored in the value

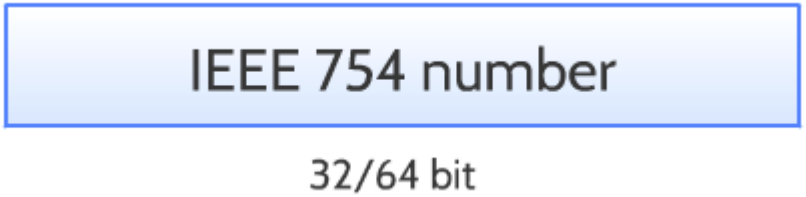


String Representation

- String is an 8 byte long value
- Several string types are supported in Jerry besides the usual character array
  - 32 bit value stored in the value field
  - Magic (frequently used) string indicies

Number representation

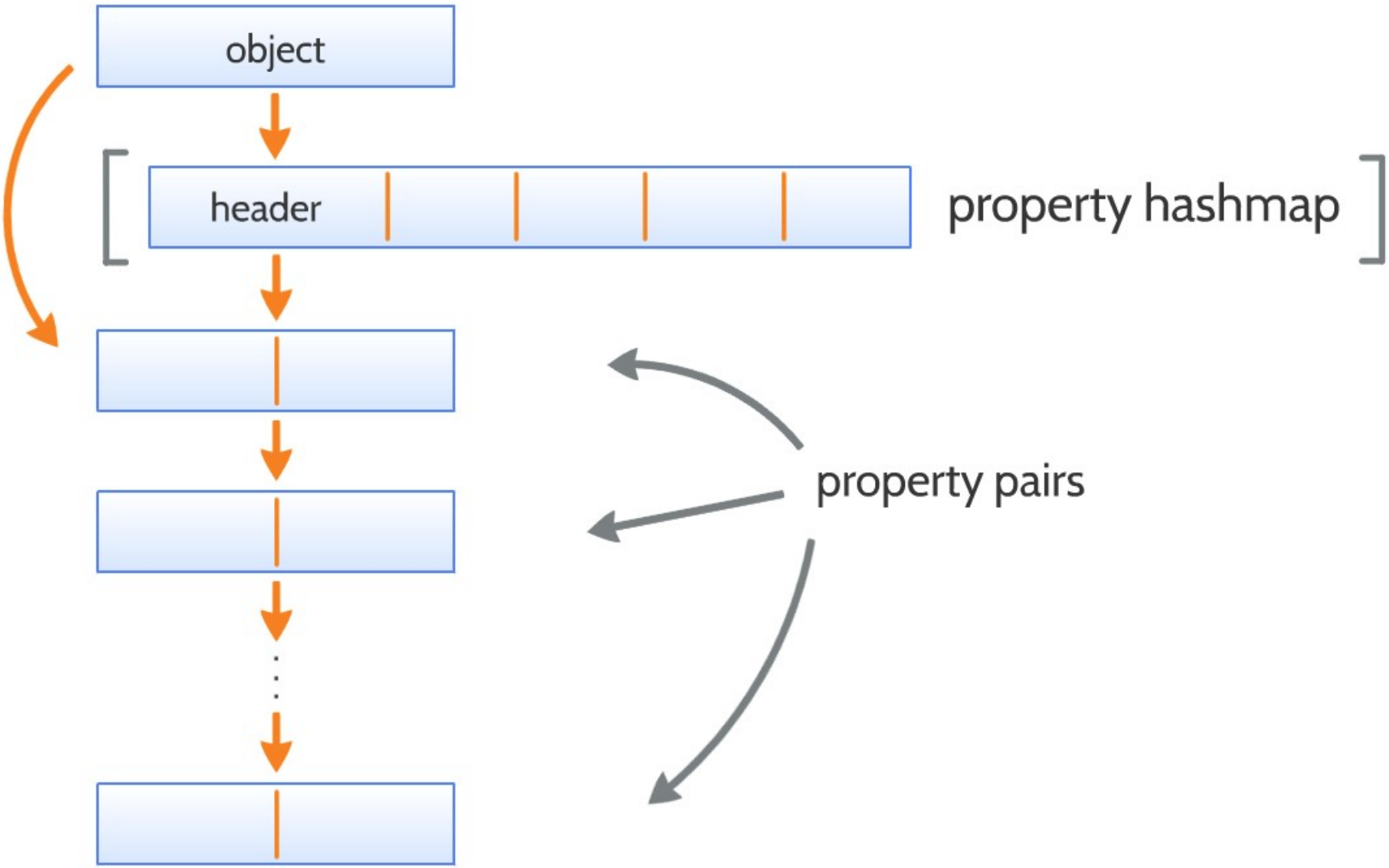
- Numbers are single or double precision values
  - Single precision numbers do not satisfy the ECMAScript requirements but provides faster computation



Object Representation

- Objects represent data objects and lexical environments
  - All functions are data objects in JavaScript
- Garbage collector can visit all existing objects
- Data objects have a property list
  - Named data, named accessor properties
  - Internal properties

Property List of a Data Object



Named Property Resolution

Named Property Resolution

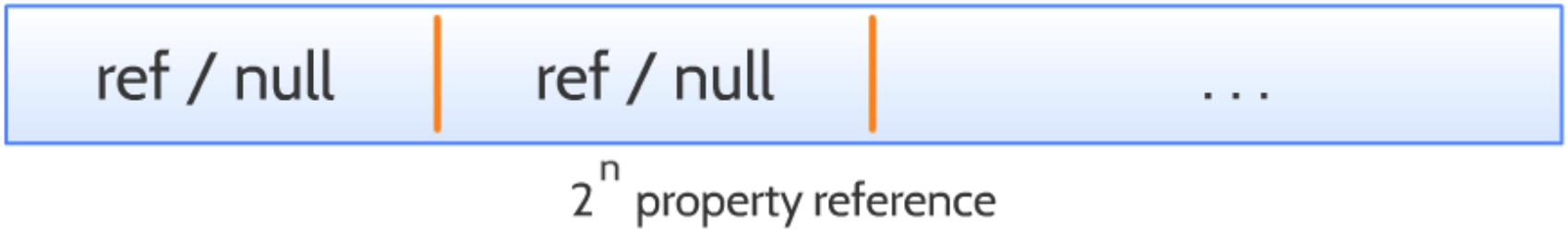
- Resolving a named property is among the most frequent ECMAScript operations
- Linear search of property list is slow
  - It hurts performance too much
- JerryScript uses two techniques to accelerate named property search
  - A global property cache called lookup cache
  - An optional property hashmap for objects with large property count

Property Lookup Cache

- A global, fixed size table where each cell contains an object, property name, and property reference triplet
- Number of rows and columns can be configured at compile time
  - Default is 256 rows and 2 columns → 4Kbyte RAM

Property Hashmap

- An array of property references belongs to a given object
- This hashmap is generated automatically for objects with large property count
- This hashmap is optional, it can be deleted by garbage collector on low-memory conditions



Property Hashmap Search Algorithm


```
// object_hashmap_p->size is a power of 2
mask = object_hashmap_p->size - 1;
index = property_name_p->hash & mask;
// step_table contains prime numbers
step = step_table[property_name_p->hash & (STEP_TABLE_SIZE - 1)];

while (object_hashmap_p->items[index] != NULL)
{
   if (string_equal (object_hashmap_p->items[index]->name_p,
                     property_name_p)
   {
     return object_hashmap_p->items[index];
   }
   index = (index + step) & mask;
}
return NULL;
```

Summary

Summary

- JerryScript is optimized for low-memory environments
  - Parser and executor requires small amount of memory
  - Small structures are used for representing ECMA values
    - Compressed pointers
- Performance matters even on embedded systems
  - Efficient named property search

Thank you.