JSJS

Language Reference Manual

—

Jain Bahul	bkj2111
Srivastav Prakhar	ps2894
Jain Ayush	aj2672
Sadekar Gaurang	gss2147

Contents

Introduction

JSJS is a strongly typed language for the web. Taking inspiration from languages such as OCaml, Scala and TypeScript, JSJS aims to be a pragmatic and a powerful language that can be used to build real-world applications. Since JSJS compiles down to Javascript, it can run both in the browser and on the server (Node.js). While designing the syntax and semantics of the language our goal has been the following -

- Minimal number of keywords
- Approachable to Javascript users
- Familiar to functional programmers
- Explicit is better than implicit

Comparison with Javascript

Literals

JS	JSJS
`3`	`3`
`3.1415`	`3.1415`
`"Hello World!"`	`"Hello World!"`
`'Hello world!'`	Cannot use single quotes for strings
`true`	`true`
`[1,2,3]`	`[1,2,3]`
`{ "foo" : "1", "bar" : "2" }`	`{ "foo" : "1", "bar" : "2" }`

Strings

JS	JSJS
`'abc' + '123'`	`"abc" ^ "123"`
`'abc'.length`	`String.length("abc")`

Maps

JS	JSJS
`x["foo"] = "1";`	`Map.set(map, "foo", "1");`
`x["bar"] = "2";`	`Map.set(map, "bar", "2");`

Functions

JS

```
function(x,y) { return x + y; }
List.map(function(x) { return x*x; }, [1,2,3,4]);
Math.max(3, 4)`
var filter = function(f, xs) { ... }
```

JSJS

```
/\(x : num, y : num) : num => x+y;
List.map((/\(x : num): num => x * x), [1,2,3,4]);
Math.max(3, 4)
val filter = /\(f: (num) -> bool, xs: list num): list num => { ... }
```

Assignments

JS	JSJS
`var x = 10`	`val x = 10`
`var x = 'foo';`	`val x = "foo";`
`var x = true;`	`val x = true;`
`var x = [1,2,3];`	`val x = [1,2,3];`
`var x = { "foo" : "1", "bar" : "2" };`	`val x = {"foo": "1","bar": "2" };`
`var x = function(x, y) { return x+y; }`	`val sq = /\(x: num): num => x+y;`

Control Flow

JS	JSJS
`if(x > y) { return x; }`	NA, else construct necessary
`if(x > y) { return x; } else { return y; }`	`if x > y then x else y`
`return 42;`	No return statements, only expressions

Types

Types are at the forefront of JSJS. Every value declaration, functions, and even compound literals need an explicit type definition. This section elaborates more on the different types of data that JSJS supports.

Primitive Types

There are four fundamental types or *primitive* types in JSJS.

Number

The `num` or the `number` type in JSJS corresponds to the Number type in Javascript.

According to the ECMAScript standard, there is only one number type: the double-precision 64-bit binary format IEEE 754 value (number between -(253 -1) and 253 -1). There is no specific type for integers and hence the same type is used to represent floating-point numbers.

AST

```
type primitiveType =
  | TNum
```

String

The `string` type in JSJS is used to represent textual data and corresponds to the String type in Javascript. It is a set of "elements" of 16-bit unsigned integer values. Each element in the String occupies a position in the String. The first element is at index 0, the next at index 1, and so on. The length of a String is the number of elements in it.

AST

```
type primitiveType =
  | TString
```

Boolean

The `bool` type in JSJS represents a logical entity and can have two values: `true`, and `false`.

AST

```
type primitiveType =
  | TBool
```

Unit

Unit, written `unit`, is a built-in type that has only one value. It is mostly used for functions that causes side effects and have no useful return value.

Unit is also used for interopability with functions in Go that has no return value at all. For example, the JSJS expression below would have the type `string -> unit` in JSJS.

```
val nothing = /\(x: string): unit => print(x)
```

The literal `unit` has the type `unit`.

AST

```
type primitiveType =
  | TBool
```

Composite Types

There are two primary composite types in JSJS

Lists

Like other functional programming languages, Lists are the fundamental data structures in JSJS. They serve as the primary basis of storing one or more related values together. The type signature of a list is `list T` where `T` is one of the other types - either primitive or composite. Here are a few examples of defining list types - - `list num` - `list string` - `list list bool`

Lists in JSJS are homogenous, i.e they can contain only one type of data. A list whose type is declared as `list num` can only store elements of `num` type. Lists can also contain other lists which have a type `list list T`.

AST

```
type primitiveType =
  | TList of primitiveType
```

Maps

Maps is another important data structure in JSJS and is treated as a first-class citizen. Since JSJS compiles down to Javascript where Maps (called objects) are used extremely liberally, Javascript programmers will feel right at home in accessing this useful data structure with as much ease as they are used to. Map types follow a different syntax for declaration: `<T: U>` where `T`is the type of the key and `U` is the type of the value stored in the Map. Example declarations of Maps are -

```
// simple maps
val names : <string: num> = { ... }
val friends : <string: list num> = { ... }

// nested maps
val people : <string: <string: bool>> = { ... }
```

Like lists, Maps in JSJS are homogenous. Like other strongly typed languages, the keys of a map should have the same type and so should the values.

AST

```
type primitiveType =
  | TMap of primitiveType * primitiveType
```

Function Types

Like other values, functions are expressions in JSJS. That means that functions also have types and functions expression also have a function type. The type of a function, is mapping of types from its formal arguments to its return type. Thus a function that takes `n` arguments has the following type signature - `(T1, T2, T3, ... Tn) -> T`.

```
// type of a function that takes two arguments
// of type num and returns a bool type.
(num, num) -> bool
```

The type of a function is determined at the time of its declaration. Each formal argument in a function definition should have a type attached followed finally by the return type of the function.

```
// a function declaration with explicit type annotations
val pow : (num, num) -> num = /\(x: num, y: num): num = {
   // ...
}
```

Function types in the JSJS compiler are implemented using mutually recursive algebraic data-types.

AST

```
type primitiveType =
  | ...
and funcType = primitiveType list * primitiveType
;;
```

Generic Types

JSJS also supports polymorphic types. Polymorphic function types can contain of type variables. These are like placeholders for the types used when applying the polymorphic function. A type variable has to be defined by an uppercase single letter.

```
val map = /\(f: T -> U, xs: list T): list U => {
 // ...
}
```

AST

```
type primitiveType =
  | T of char
```

Type Declarations

Type annotations of all expressions (except functions) are optional. The type system *infers* the type of the expression from the expression on the right and assigns that type to the val on left.

Grammar

```
assigns:
  | VAL ID COLON primitive ASSIGN expr       { Assign($2, $4, $6) }
  | VAL ID ASSIGN expr                       { Assign($2, TSome, $4) }
```

AST

```
type expr =
  | Assign of string * primitiveType * expr
```

Examples

```
// explicit type definition
val name : string = "Foobar";

// types are optional and count is assigned the `num` type.
val count = 10 + 20;

// wrong type annotations will raise type mismatch errors
val happy? : bool = "string1" ^ " " ^ "string2";

// functions need an explicit type declaration
val square = /\(x: num, y: num): num => x * y;
```

Types in AST

In conclusion, the types of JSJS are defined in the AST as below -

```
type primitiveType =
  | T of char
  | TSome
  | TNum
  | TString
  | TBool
  | TUnit
  | TFun of funcType
  | TList of primitiveType
  | TMap of primitiveType * primitiveType
and funcType = primitiveType list * primitiveType
;;
```

Lexical Conventions

Comments

Only single-line comments are allowed in JSJS. Anything followed by `//` on the line will be considered as a comment and will be ignored by the compiler.

Lexer:

```
rule token =
    parse
    | "//"                        { comment lexbuf; }

and comment =
    parse
    | '\n'                        { token lexbuf }
    | _                           { comment lexbuf }
```

Example:

```
// This is a comment

......... // This is a comment too
```

Identifiers

Identifiers are sequences of characters used for naming JSJS entities. All identifiers cannot have the same spelling (character sequence) as a JSJS keyword, JS keyword, or a boolean literals, or a compile-time error occurs. Lowercase letters and uppercase letter are distinct, such as `isEmpty?` and `isempty?` are two different identifiers.

Value and Function Identifiers

Valid identifier characters for values and functions include ASCII letters, decimal digits, underscore character and the '?' character. The first character must be a small case alphabetic character. The '?' character can only be used as the last character of the identifier.

Regular Expression:

```
id = ['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* ['?']?
```

Example:

```
// Valid Identifiers for values and functions
x
age
totalAmount
total_amount
isEmpty?
person1

// Invalid Identifiers for values and functions
X
1person
isEmpt?y
&name
Person
```

Module Identifiers

Valid identifier characters for modules include only ASCII letters. The first character must be an upper case alphabetic character.

Regular Expression:

```
module = ['A'-'Z'] ['a'-'z' 'A'-'Z']+
```

Example:

```
// Valid Identifiers for modules
List
StringMap
HashSet

// Invalid Identifiers for modules
stringMap
hash_set
&list
```

Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose. JSJS recognizes the following keywords:

```
val, if, then, else, num, bool, string, unit, true, false, list
```

Separators

A separator separates tokens. Separators themselves are simply single-character tokens.

Character	Token
`'('`	`{ LPAREN }`
`')'`	`{ RPAREN }`
`'{'`	`{ LBRACE }`
`'}'`	`{ RBRACE }`
`'['`	`{ LSQUARE }`
`']'`	`{ RSQUARE }`
`';'`	`{ SEMICOLON }`
`','`	`{ COMMA }`
`'.'`	`{ DOT }`

Literals

A literal is a source code representation of a value of a primitive type or a composite type.

Number Literals

A number literal has the following parts: a whole-number part, an optional decimal point (represented by an ASCII period character), and a following fraction part. The whole number and fraction parts are defined by a single digit 0 or one digit from 1-9 followed by more ASCII digits from 0 to 9.

Regular Expression:

```
number = digit+ '.'? digit*
```

Example:

```
// Valid number literals
4
4.5
0.0002
42.

// Invalid number literals
.7
1e+3
```

Boolean Literals

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters.

```
bool = "true" | "false"
```

String Literals

A string literal is represented as a sequence of zero or more ASCII characters enclosed in two double quotes. The following characters are represented with an escape sequence, which consists of a backslash and another character:

- “\” - backslash
- “"” - double-quote
- “\n” - new line
- “\r” - carriage return
- “\t” - tab character

Regular Expression:

```
string = ([' '-'!' '#'-'[' ']'-'~'] | '\\' ['\\' '"' 'n' 'r' 't'])*
```

List Literals

A list literal is represented by comma separated expressions that evaluate to literals of the same type enclosed within square brackets.

Grammar:

```
| LSQUARE actuals_opt RSQUARE              { ListLit($2) }

actuals_opt:
    | opts = separated_list(COMMA, expr)       { opts }
```

Example:

```
// Literal for a list of numbers
[1,2,3,4,5,6]

// Literal for a list of strings
["jsjs", "is", "awesome", "!!"]

// Literal for a list of bools
[1 == 1, 2 == 3, 5 <= 4, !true]
```

Map Literals

A map literal is represented by comma separated key-value pairs that are enclosed within curly braces. A key can only be expressions of number, string or a bool type, while values can be expressions of any type. A key-value pair is written as `<key> : <value>`.

Grammar:

```
 | LBRACE kv_pairs RBRACE                           { MapLit($2) }

kv_pairs:
    | kv = separated_list(COMMA, kv_pair)      { kv }

kv_pair:
    | expr COLON expr                               { $1, $3 }
```

Example:

```
// A map literal with key as a number and value a string.
{ 1: "One", 2: "Two", 3: "Three", 4: "Four" }

// A map literal with key as a number and value as a list of strings.
{ 1: ["One", "Uno"], 2: ["Two", "Dos"], 3: ["Three", "Tres"] }
```

Operators

The following operators are reserved lexical elements in the language. See the expression and operators section for more detail on their defined behavior

Character	Token
`'+'`	`{ PLUS }`
`'-'`	`{ MINUS }`
`'*'`	`{ MULTIPLY }`
`'/'`	`{ DIVIDE }`
`'%'`	`{ MODULUS }`
`'^'`	`{ CARET }`
`'<'`	`{ LT }`
`'<='`	`{ LTE }`
`'>'`	`{ GT }`
`'>='`	`{ GTE }`
`'=='`	`{ EQUALS }`
`'='`	`{ ASSIGN }`
`'!'`	`{ NOT }`
`'&&'`	`{ AND }`
`'||'`	`{ OR }`

Functions

All functions in JSJS are Lambda expressions. Since functions are treated as first class citizens, these lambda expressions can be assigned as values to identifiers, passed as arguments to other functions, and returned as values from other functions.
To make a named function declaration, a lambda expression is assigned to an identifier using the `val` keyword, just like any other type declaration.

Lambdas

Lambda expressions are denoted by the symbol `/\`, which resembles the upper case Greek letter Lambda. JSJS Lambda expressions have the following form:

```
/\(argument declarations if any) : return type => {
  Block of expressions, the last of which
  is the value that is returned
}
```

A shorthand syntax is also supported if the body of the Lambda is a single statement:

```
/\(argument declarations, if any) : return type => expression
```

The argument declarations *must* be annotated with their types, and a return type of the body of the `/\` expression also *must* be specified.
For example, the following / expression takes a single number `x` as an argument and evaluates the square of `x`.

```
/\(x : num) : num => x * x
```

It is also possible to define Lambda expressions that don't take any arguments:

```
/\() : unit => println("hello world")
```

or those that take multiple arguments:

```
/\(fname : string, lname : string) : string => "Hello " ^ " " ^ fname ^ " " ^ lname
```

The body of a Lambda can also be a block of expressions, where the last expression is the one that is implicitly evaluated and returned. The following `/\` takes a single numeric argument `x` and adds the value `y` - assigned as 10 in the body - to `x`.

```
/\(x : num) : num => {
  val y = 10;
  x + y;
}
```

AST for /\ expressions:

```
type expr:
| FunLit of func_decl

func_decl = {
  formals     : (string * primitiveType) list;
  return_type : primitiveType;
  body        : expr;
}
```

Grammar:

```
literals:
| LAMBDA LPAREN formals_opt RPAREN COLON primitive FATARROW expr %prec ANON {
        FunLit({
            formals = $3; return_type = $6; body = Block([$8]);
        })
    }
| LAMBDA LPAREN formals_opt RPAREN COLON primitive FATARROW block {
        FunLit({
            formals = $3; return_type = $6; body = Block($8);
        })
    }
```

Function Types

When an identifier is assigned to a /\ expression, it becomes a value of the function type. The type of a function is

```
fn : (A,B) -> C
```

Here, `fn` takes 2 arguments of type `A` and `B` respectively, and evaluates an expression or block of type `C`. In general, a function type is a list of input argument types (optional) and return type.

When assigning an identifier to a /\ expression using the `val` keyword, annotating the identifier with the function type is optional, just like type specifiers for all other expressions.

```
// function type explicitly specified
val cube : (num) -> num = /\(x : num) : num => x * x * x ;

// function type of val not annotated
val whatDoYouKnow = /\(name : string) : string => {
  if name == "John Snow"
  then "Nothing"
  else "Something" ;
} ;
```

AST

```
type primitiveType =
| TFun of funcType
and funcType = primitiveType list * primitiveType
```

Grammar

```
primitive:
| LPAREN args RPAREN THINARROW primitive  { TFun($2, $5) }

args:
| args = separated_list(COMMA, primitive) { args }
```

Function Calls and Usage

Functions are called by invoking the name of the function and passing it actual arguments. These arguments can be any kind of expressions.
Every function call is itself an expression, and evaluates to a value which has a type (the return type of function).

```
// function called with a numeric literal
val sq5 = sq(5)

// function called with an expression
val x = 8;
val y : num = sq(x);
val z = cube(x + y);

// function called with a function as an argument
val addOne = /\(x : num) : num => x + 1;
val addOneSqr = /\(f: (num) -> num, g: (num) -> num, x : num) : num => f(g(x));
val result = addOneSqr(sq, addOne, 8);
```

AST

```
type expr =
| Call of string * expr list
```

Grammar

```
expr:
| ID LPAREN actuals_opt RPAREN              { Call($1, $3) }
```

Operators

JSJS supports various operators for different data types. Broadly, JSJS includes Arithmetic Operators, Relational Operators, Boolean operators, Assignment Operator and String Operator. While most of these are binary operators, some are unary.

```
| Binop of expr * op * expr
| Unop of op * expr
```

The above code excerpt defines two types of expressions. The former defines a binary operator while the latter gives the format of a unary operator.

Arithmetic Operators

JSJS supports the following arithmetic operators: `+`, `-`, `*`, `/`, `%`.. All arithmetic operators require two operands of `num` data types. These can be either literals or variables or a combination of the two.

1. Addition

```
10 + 7; //17
x + y;
```

2. Subtraction

```
10 - 7; //3
x - y;
```

3. Multiplication

```
10 * 7; //70
x * y;
```

4. Division

```
21 / 7; //3
x / y;
```

5. Modulus

```
10 % 7; //3
x % y;
```

Relational Operators

The following relational operators are supported by JSJS: `==`, `!=`, `>=`, `<=`, `>`, `<`. Relational operators require two operands. These operands can be of any primitive type namely, int, bool, string or unit. The only condition is that both the operands should be of the same type. These expressions always return a value of boolean data type.

1. Equals

```
5 == 5 //true
"abc" == "def" //false
true == false //false
```

2. Not Equals

```
5 != 5 //false
"abc" != "def" //true
true != false //true
```

3. Less than

```
5 < 5 // false
"abc" < "def" //true
true < false //false
```

4. Less than or Equals

```
5 <= 5 //true
"abc" <= "def" //true
true <= false //false
```

5. Greater than

```
5 > 5 //false
"abc" > "def" //false
true > false //true
```

6. Greater than or Equals

```
5 >= 5 //true
"abc" >= "def" //false
true >= false //true
```

Boolean operators

JSJS supports three boolean operators: `&&, ||, !.&&and||act on two operands whereas¡` is a unary operator. These act on boolean data types and return a single value of type boolean.

1. And

   ```
   true && false //false
   x && y
   ```

2. Or

   ```
   true || false //true
   x || y
   ```

3. And

   ```
   !true //false
   !x
   ```

Assignment Operator

The assignment operator is used to assign a value to an identifier. The value of the expression on the right side is evaluated and assigned to the identifier on the left hand side.

```
val x : num = 5 + 3;
val y : string = "abc" ^ "def";
val z : bool = true;
```

String operator

JSJS includes an operator for strings as well. The ˆ operator is the string concatenation operator, which takes two strings and returns an output string formed by the concatenation of the two.

```
"abc" ^ "def" //"abcdef"
x ^ y
```

Operator precedence

JSJS defines a precedence order in which operations are performed when more than one operators are present in a single expression. The operators sharing the same precedence are evaluated according to their associativity. Operators which are left associative are evaluated from left to right. Similarly, right associative operators are evaluated from right to left. In JSJS, all operators are left associative except for the assign(=) operator. Following is the chart of operator precedence.

```
*, /, %
+, -
<=, >=, <, >, ==, !=
!
^, &&, ||
=
```

The following OCaml code defines the operator precedence along with their associativity for our parser. The precedence increases from top to bottom which means that operators on the bottom are always evaluated first.

```
%right ASSIGN
%left CARET AND OR
%left NOT
%left LTE GTE LT GT EQUALS NEQ
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULUS
%left NEG
```

Expressions

Everything in JSJS is an expression. Accordingly, an entire JSJS program can be defined as a list of expressions.

```
program:
    | expr_list EOF                          { $1 }

expr_list:
    | exprs = nonempty_list(delimited_expr)   { exprs }
```

All these expressions are composed of identifiers, operators, literals and function calls. Following is an exhaustive list of different types of expressions in JSJS:

```
type expr =
  | Binop of expr * op * expr
  | Unop of op * expr
  | NumLit of float
  | BoolLit of bool
  | StrLit of string
  | MapLit of (expr * expr) list
  | ListLit of expr list
  | Assign of string * primitiveType * expr
  | Val of string
  | Block of expr list
  | If of expr * expr * expr
  | Call of string * expr list
  | FunLit of func_decl
  | ModuleLit of string * expr
and
func_decl = {
  formals     : (string * primitiveType) list;
  return_type : primitiveType;
  body        : expr;
};;
```

Blocks

A block is a list of several expressions enclosed in curly braces. The entire block is executed in the order in which expressions appear and the value of the last expression is returned. In JSJS, blocks are encountered inside of if-then-else statements and function definitions.

```
block:
    | LBRACE expr_list RBRACE                { $2 }

expr_list:
    | exprs = nonempty_list(delimited_expr)   { exprs }

delimited_expr:
    | expr SEMICOLON                          { $1 }
```

if-then-else

if-then-else behaves similarly to the standard control flow constructs of programming languages. Following is the grammar definition of an if-then-else statement.

Grammar:

```
| if expr then block else block     { If($2, Block($4), Block($6)) }
```

One important point to keep in mind is that the return type of then and else blocks should be same. Otherwise the compiler will throw a type mismatch error.

Example:

```
if x > y
then { print(x); x; }
else { print(y); y; }
```

The curly braces in then and else blocks are required only if the blocks consist of more than one expression but are optional otherwise. They are accordingly handled in the grammar as well.

Standard Library Functions

JSJS provides a simple set of standard library functions to perform basic operations on lists, maps and work with standard input/output. By default, all the following libraries are automatically available.

List Module

The awesome `List` module provides essential functions to do basic operations on lists:

1. `List.hd`: Returns the head of the list.
2. `List.tl`: Returns a list without the head of the original list.
3. `List.cons`: Appends an element to the head of the list.
4. `List.length`: Returns the total number of elements in the list.
5. `List.nth`: Returns the nth element of the list.
6. `List.contains?`: Return true if a given element exists in the list, else returns false.
7. `List.isEmpty?`: Return true if the list does not contain any elements, else returns false.
8. `List.rev`: Reverses the given list.

Map Module

The awesome `Map` module provides essential functions to do basic operations on maps:

1. `Map.set`: Adds an entry in a map.
2. `Map.get`: Returns a value given the key from a map.
3. `Map.remove`: Returns a map after removing key from a map.
4. `Map.clear`: Removes all entries of the map.

String Module

The `String` module is provided to assist users with string manipulation:

1. `String.length`: Returns the length of the string
2. `String.concat`: Concatenates two strings
3. `String.substring`: Returns a substring of the given string

IO Module

The `IO` module is used to interact with the standard IO

1. `IO.print`: Display on standard output
2. `IO.println`: Display on standard output, followed by a new line character.
3. `IO.read`: Reads contents from standard input.