CellScript Language Specification
Draft v0.1.1

WARNING: This is a draft of the CellScript Language Specification. This is not an accepted standard. Every syntax defined in this document is subject to change without prior notice.

This document is an extension of the ECMA-262 5.1th Edition, ECMAScript Language Specification. It also builds on segments from the ECMA-262 6th Edition Draft: Symbols, Proxies, Generators, Default Function Parameters…

Language Overview

To define the CellScript language we should start with the CellScript Compiler. Basically this is a JavaScript to JavaScript compiler based on the ECMA-262 ECMAScript 5.1th Edition Specification. What makes the compiler useful is it’s support for Compiler Modules. These modules are extensions of the compiler with added, extended or replaced language grammar and compiler features. The CellScript Language is built from several Compiler Modules.

These modules are the following:
- JavaScript+ Module
- InqScript Module
- CellScript Module
- Experimental Features Module

JavaScript+ Module
This module extends the compiler with support for several ECMAScript 6.0 and 7.0 features. This features follow or build on the new standard, but compile into code compatible with the original standard.

InqScript Module
This module adds support for Task-based Asynchronous Programming (TAP) pattern with a C#-like implementation of async and await statements and expressions.

CellScript Module
This module extends the language with a contract based abstract programming pattern for developing applications for the Cell Runtime.

Experimental Features Module
This module adds in development features to the compiler, whose can be enabled with a compiler flag.

1. Low-level Additions

1.1. Pre-processing Directives

```
PreprocessingDirective :
      # PreprocessingDirectiveElement

PreprocessingDirectiveElement :
      PreprocessingFlag BooleanLiteral
      PreprocessingVariable Literal[except NullLiteral]
      PreprocessingOperation
      cell StringLiteral
      region IdentifierName

PreprocessingFlag : one of
      typechecks  strict     minify

PreprocessingVariable : one of
      define      undef      constant    const
      ifdef       ifndef

PreprocessingOperation : one of
      else        endif      endregion
```

1.2. Literals

1.2.1. TimeLiteral

Represents an amount of time in milliseconds (ms), seconds (s), minutes (m), hours (h), days (d) or weeks (w).

Compilation:
A TimeLiteral compiles into a NumericLiteral which will be the millisecond representation of the TimeLiteral value.

```
TimeLiteral :
      NumericLiteral ms
      NumericLiteral s
      NumericLiteral m
      NumericLiteral h
      NumericLiteral d
      NumericLiteral w

Literal ::
      NullLiteral
      BooleanLiteral
      NumericLiteral
      StringLiteral
      RegularExpressionLiteral
      TimeLiteral
```

1.3. Formal Parameter List

TODO

```
FormalParameterList :
      FormalParameter
      FormalParameter , FormalParameterList

FormalParameter :
      IdentifierName FormalParameterTail_opt
      RestParameter FormalParameterTail_opt

FormalParemeterTail :
      DefaultValue
      TypeSpecification

DefaultValue :
      = LeftHandSideExpression
      = ConditionalExpression

TypeSpecification :
      : IdentifierName

RestParameter :
      . . . IdentifierName
```

1.4. Let Statement

```
LetStatement :
      let IdentifierName
      let IdentifierName = Expression
```

1.5. Foreach and For…of Statement

```
ForeachStatement :
      foreach ( IdentifierName in Expression ) Statement
      foreach ( let IdentifierName in Expression ) Statement
      foreach ( var IdentifierName in Expression ) Statement

ForOfStatement :
      for ( IdentifierName of Expression ) Statement
      for ( let IdentifierName of Expression ) Statement
      for ( var IdentifierName of Expression ) Statement
```

2. Task-based Asynchronous Pattern (TAP)

The TAP implementation is based on the C# implementation of the same pattern. The CellScript implementation supports both NodeJS Style Asynchronous Programming (callbacks) and the Promise pattern. Other patterns can be also adapted to TAP with minimal work.

2.1. Async

TAP introduces the **async** keyword and the **async** modifier. Both declares an execution scope where task-based asynchronous operations are supported. This scope is called **Awaited Scope**.

2.1.1. Async Modifier

The Async Modifier identifies a Function Expression or a Function Declaration whose body is an Awaited Scope.

Compilation:

As TAP is a state machine based approach and JavaScript (at least in ECMA6) already has a state machine implementation, called Generator Function, it is logical to compile functions with Awaited Scopes into Generator Functions. Also Generator Functions declared in CellScript are compiled as functions with Awaited Scope.

The Async Modifier is an extension of the Function Declaration and the Function Expression grammar of ECMAScript Version 6.0:

```
FunctionDeclaration :
      function asyncopt Identifier ( FormalParameterListopt ) { FunctionBody }
      function * Identifier ( FormalParameterListopt ) { FunctionBody }

FunctionExpression :
      function asyncopt Identifieropt ( FormalParameterListopt ) { FunctionBody }
      function * Identifieropt ( FormalParameterListopt ) { FunctionBody }
```

2.1.2. Async Keyword

The Async Keyword is a required addition for the Async Statement, described in 2.1.3.

```
Keyword :: one of
      break       do          instanceof  typeof
      case        else        new         var
      catch       finally     return      void
      continue    for         switch      while
      debugger    function    this        with
      default     if          throw
      delete      in          try
      class       module      extends
      self        private
      yield       async
```

2.1.3. Async Statement

While functions (and methods) can hold an Awaited Scope using the Async Modifier, there is still a need for a construction which enables calling of a function with an Awaited Scope or an Awaitable Function (described in section 2.2.2) using TAP. The Async Statement is the CellScript construction for this. Also this construction is required addition of CellScript to the original TAP (described on MSDN) as in JavaScript functions with an Awaited Scope cannot be properly called outside of an Awaited Scope.

Compilation:

An Async Statement defines a Promise with Awaited Scope, which is implemented in the inq-core NPM module. As of this, the compilation wraps the code into an InqPromise.

The Inq Statement imports the inq-core nom module and enables Inq related features.

InqStatement :
    inq ;

AsyncStatement :
    async *Expression* **;**
    async *Expression Catch Finally*$_{opt}$;
    async *IdentifierName*$_{opt}$ *Block*
    async *IdentifierName*$_{opt}$ *Block Catch Finally*$_{opt}$

The Abort Statement terminates an async statement. On JavaScript level it rejects the InqPromise.

AbortStatement :
    abort *IdentifierName* **;**

2.2. Await and Task Expressions
TAP is based on asynchronously executable Tasks. This (as in the original implementation too) is separated into two parts: tasks and execution. Task Expressions define a task which can be stored for later execution, while Await Expressions execute an inline or stored task using TAP.

2.2.1. Task Expressions
A task define an asynchronously executable operation which can be stored in a variable before executing with an await expression (described in section 2.2.2). In CellScript Task Expressions implement tasks.

Compilation:
Task expressions are be “inqified” during compilation: These expressions are compiled into chained calls to the Inq API.

UnaryExpression :
    PostfixExpression
    delete *UnaryExpression*
    void *UnaryExpression*
    typeof *UnaryExpression*
    ++ *UnaryExpression*
    -- *UnaryExpression*
    + *UnaryExpression*
    - *UnaryExpression*
    ~ *UnaryExpression*
    ! *UnaryExpression*
    => *UnaryExpression*
    -> *UnaryExpression*
    ***>** *UnaryExpression*
    ***=>** *UnaryExpression*
    ***->** *UnaryExpression*
    task *TaskBinding*$_{opt}$ *TaskModifier*$_{opt}$ *UnaryExpression*
    task parallel *ParallelTaskModifier*$_{opt}$ *UnaryExpression*

The Timeout Modifier enables setting the maximum execution time for a task. After this time the task (promise) gets rejected.

TimeoutModifier :
    timeout *AssignmentExpression*

The Concurrent Modifier enables setting the maximum count of operations running at the same time for a parallel task.

ConcurrentModifier :
    concurrent *AssignmentExpression*

The Fallback Modifier enables setting a default result for a task which will be provided automatically if the task fails or no result is provided.

FallbackModifier :
    fallback *AssignmentExpression*

The Repeat Modifier enables setting how many times the task should be executed before completing.

RepeatModifier :
    repeat *AssignmentExpression*

The Retry Modifier enables setting how many times the task should be retried on failure. Also a back off time can be provided for the modifier. (For more information refer to the Inq Documentation)

RetryModifier :
    retry *AssignmentExpression*
    retry *AssignmentExpression* **backoff** *AssignmentExpression*

2.2.2. Await Expression

As in JavaScript there are two frequently used patterns for asynchronous programming, CellScript's implementation of TAP supports both. These are Promises and Callbacks. While Promises (based on Promises/A specification, aka objects with a then method) are covered by Tasks, Callbacks require more explanation.

Callbacks chains are very old and well known ways of async programming in JavaScript and NodeJS also built on this foundation. But there are several different ways of using callbacks, errors and results can be provided in different orders and ways.

So the implementation of CellScript ensures that most implementation supported out of the box, while rare implementations require only a small adaptation (a wrapper function to transform the callback). Functions using the supported formats are called Awaitable Functions.

When calling an Awaitable Function in an Await Expression there are two possibilities:

A. The function uses the NodeJS Style Callback Pattern (the callback is the last argument of the function and uses the following signature: function (error, result)) In this case the await expression automatically handles the callback.
B. The function provides a separate error callback and a result callback in the argument list. In this case the Error Callback Sign (**#**) and the Result Callback Sign (**%**) are required to identify the position of the callbacks in the argument list.

<u>Compilation</u>:
Await expressions are be "inqified" during compilation: These expressions are compiled into chained calls to the Inq API.

```
UnaryExpression :
      PostfixExpression
      delete UnaryExpression
      void UnaryExpression
      typeof UnaryExpression
      ++ UnaryExpression
      -- UnaryExpression
      + UnaryExpression
      - UnaryExpression
      ~ UnaryExpression
      ! UnaryExpression
      task TaskBinding_opt TaskModifier_opt UnaryExpression
      task parallel ParallelTaskModifier_opt UnaryExpression
      await TaskBinding_opt TaskModifier_opt UnaryExpression
      await* TaskModifier_opt UnaryExpression
      await parallel ParallelTaskModifier_opt UnaryExpression
      wait UnaryExpression
      bound UnaryExpression

CallExpression :
      MemberExpression Arguments
      CallExpression Arguments
      CallExpression [ Expression ]
      CallExpression . IdentifierName

Arguments :
      ( )
      ( ArgumentList )

ArgumentList :
      CallbackSign
      AssignmentExpression
      ArgumentList , AssignmentExpression

CallbackSign :: one of
      #     %
```

3. Arrow Functions

Arrow functions provide…

```
AssignmentOperator : one of
      *=    /=    %=    +=    -=    <<=   >>=   >>>=  &=    ^=    |=    *>    =>
      ->    *=>   *->

UnaryExpression :
      PostfixExpression
      delete UnaryExpression
      void UnaryExpression
      typeof UnaryExpression
      ++ UnaryExpression
      -- UnaryExpression
      + UnaryExpression
      - UnaryExpression
      ~ UnaryExpression
      ! UnaryExpression
      await TaskBinding_opt TaskModifier_opt UnaryExpression
      await* TaskModifier_opt UnaryExpression
      await parallel ParallelTaskModifier_opt UnaryExpression
      task TaskBinding_opt TaskModifier_opt UnaryExpression
      task parallel ParallelTaskModifier_opt UnaryExpression
      wait UnaryExpression
      bound UnaryExpression
      => UnaryExpression
      -> UnaryExpression
      *> UnaryExpression
      *=> UnaryExpression
      *-> UnaryExpression
```

4. Cells

CellScript introduces Cells, a new pattern for developing modular connected applications. Cells are executable primary modules built from Components and backed by Contracts.

4.1. Cell Declaration

A Cell is declared using a Cell Statement. This statement creates a Cell Scope, which enables other Cell related Statements and Expressions. Also the Cell Statement automatically sets up the Communication Contract.

```
CellStatement :
      cell { CellElementsopt }

CellElements :
      CellElement
      CellElement CellElements

CellElement :
      SourceElement
      EndpointStatement
      ListenStatement
      RequireStatement
      IncludeStatement
```

4.1.1. Communication Contract

The Communication Contract defines Endpoints to connect applications which each other. Also these can be used to communicate with users and define User Interfaces for Web Applications.

```
EndpointStatement :
      endpoint asyncopt HttpVerb StringLiteral authorizeopt Block

HttpVerb : one of
      get   post  put   delete
```

The Respond Statement lets an Endpoint respond to requests using any data.

```
RespondStatement :
      respond Expression
```

The listen statement initialises the Cell and enables communication with it.

```
ListenStatement :
      listen Expression
```

4.1.2. Components

4.1.2.1. Require Statement

The Require Statement is a syntactic sugar over the standard require() implementation of NodeJS or AMD.

Compilation:
The Require Statement compiles to a Variable Definition and a call to the require function. When the **as** modifier is not used the variable name is generated automatically using the following algorithm:

- if the module name is an IdentifierName, then use the module name as variable name
- if the module name is a path, use the part of the path after the last '/' sign
- automatically convert illegal characters to legal characters

```
RequireStatement :
      require RequireElementList

RequireElementList :
      RequireElement
      RequireElement , RequireElementList

RequireElement :
      StringLiteral
      IdentifierName
      StringLiteral as Identifier
      Identifier as Identifier
```

4.1.2.2. Include Statement

TODO

```
IncludeStatement :
      include silent_opt StringLiteral
```

4.1.2.3. Component Statement

TODO

```
ComponentStatement :
      component { ComponentElements_opt }

ComponentElements :
      ComponentElement
      ComponentElement ComponentElements

ComponentElement :
      CellElement
      UseStatement
```

4.1.2.4. Use Statement

TODO

```
UseStatement :
      use UseElementList

UseElementList :
      UseElement
      UseElement , UseElementList


UseElement :
      Identifier
```

5. Object Oriented Constructions

5.1. Classes

JavaScript was originally designed as an object oriented programming language and with ECMAScript 6.0 and the new **class** keyword now it'd getting closer to first class OO languages like C# or Java. CellScripts while adding support for classes in ECMAScript 5.1 based runtimes also builds on the foundations of the new JavaScript standard and the ECMAScript Next proposals: It introduces private scopes and C#-like property definitions.

Compilation:
Classes gets compiled to constructor functions and prototype inheritance as it would be done in ECMAScript 5. As added features require the inq-oo NPM module, it is also added to the compiler output.

ClassStatement :
  class *IdentifierName* *ClassTail*$_{opt}$ **{** *ClassElements*$_{opt}$ **}**

ClassTail :
  extends *IdentifierName*
  : *IdentifierName*

ClassElements :
  ClassElement
  ClassElement ClassElements

ClassElement :
  ConstructorStatement
  MethodStatement
  PropertyStatement

5.1.1. Methods

ConstructorStatement :
  constructor (*FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}**

MethodStatement :
  AccessModifier$_{opt}$ *IdentifierName* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}**

5.1.2. Properties

PropertyStatement :
  AccessModifier$_{opt}$ *IdentifierName* **{** *PropertyGetter* **}**
  AccessModifier$_{opt}$ *IdentifierName* **{** *PropertySetter* **}**
  AccessModifier$_{opt}$ *IdentifierName* **{** *PropertyGetter PropertySetter* **}**
  AccessModifier$_{opt}$ **get** *IdentifierName* **( ) {** *FunctionBody* **}**
  AccessModifier$_{opt}$ **set** *IdentifierName* **(** *IdentifierName*$_{opt}$ **) {** *FunctionBody* **}**

PropertyGetter :
  get { *FunctionBody* **}**
  get *IdentifierName*

PropertySetter :
  set { *FunctionBody* **}**
  set *IdentifierName*

5.1.3. Access Modifiers

AccessModifier : **one of**
  public **private** **static**

5.2. Modules

TODO

ModuleStatement :
  module *IdentifierName*$_{opt}$ **{** *ModuleBody* **}**

ModuleBody :
  FunctionBody
  export *Expression*
  import *Expression*