

# Complétion d'arguments

Par Laurent Dardenne, le 04/07/2017.



Niveau		
Débutant	Avancé	Confirmé
	<input type="checkbox"/>	

Conçu avec Powershell v5.1 Windows 7 64 bits.

Site de l'auteur : <http://laurent-dardenne.developpez.com/>

# Chapitres

<b>1</b>	<b>COMPLETION.....</b>	<b>3</b>
1.1	COMPLETION STATIQUE .....	3
1.2	COMPLETION DYNAMIQUE .....	4
<b>2</b>	<b>REGISTER-ARGUMENTCOMPLETER .....</b>	<b>5</b>
2.1	DETAIL DU TYPE RESULTAT : COMPLETIONRESULT .....	7
2.2	DECLARATION UNIQUE POUR DE MULTIPLE COMMANDES .....	8
<b>3</b>	<b>L'ATTRIBUT [ARGUMENTCOMPLETER()] .....</b>	<b>9</b>
3.1	PARAMETRER AVEC UN SCRIPTBLOCK.....	9
3.2	PARAMETRER AVEC UN TYPE .....	10
3.3	COMBINAISON D'ATTRIBUTS .....	11
3.4	MASQUER UN PARAMETRE .....	11
<b>4</b>	<b>COMPLETION DE COMMANDES NATIVES.....</b>	<b>12</b>
<b>5</b>	<b>INFERENCE DE TYPE.....</b>	<b>13</b>
5.1	L'ATTRIBUT OUTPUTTYPE.....	13
5.2	CONTEXTE DE COMPLETION DANS PSREADLINE.....	13
5.3	FAKEBOUNDPARAMETER .....	14
<b>6</b>	<b>LES FONCTIONS DE HOOK .....</b>	<b>16</b>
6.1	TABEXPANSION .....	16
6.2	LECTURE DE LA SAISIE UTILISATEUR.....	16
6.3	RESOLUTION DE COMMANDE.....	17

# 1 Complétion

La complétion est une aide à la saisie que l'on trouve dans un EDI, dans des [composants graphique](#) utilisées dans les GUI ou encore dans un shell en ligne de commande.

Par défaut Powershell propose une complétion sommaire, par exemple sur les paramètres d'un cmdlet. Dans le host la saisie du caractère '-' suivi de la touche *TAB* propose les paramètres les uns à la suite des autres, du premier au dernier, la combinaison des touches *Shift-TAB* les propose du dernier au premier.

Le mot anglais *completion* peut être traduit ici par 'achèvement', l'objectif est d'ajouter les caractères manquant à un début de mot ou de proposer une liste de choix dans un contexte donné, par exemple toutes les propriétés d'un objet ou encore tous les espaces de nom d'assemblies.

## 1.1 Complétion statique

Une autre aide à la saisie concerne l'argument du paramètre, dans l'exemple suivant on restreint les valeurs possibles :

```
Function Test{
    Param(
        [ValidateSet("HKCR","HKCU","HKLM","HKCC")]
        [String] $Hive
    )
    'Test'
}
```

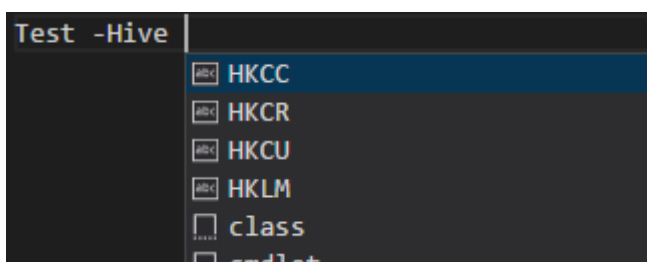
Dans ce cas et une fois avoir saisi 'Test -Hive ', la saisie supplémentaire du caractère *espace* fait que le host Powershell déclenche l'affichage des valeurs possible pour ce paramètre afin de compléter la commande.

Avec le module PSReadline la saisie la combinaison de touches *control-espace* affiche toutes les valeurs possibles :



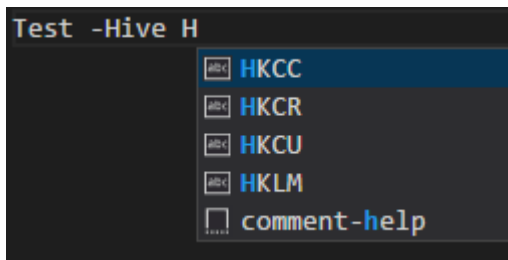
```
[STA] C:\temp> test -Hive HKCC_
HKCC HKCR HKCU HKLM
```

La complétion dans l'éditeur de Visual Studio Code est différente de celle d'une console, les valeurs possibles de l'argument sont insérées en début de liste et celle-ci contient des valeurs qui ne sont pas issues du code de la fonction :



```
Test -Hive
HKCC
HKCR
HKCU
HKLM
class
cmdlet
```

Si on saisit la lettre 'H' les valeurs proposées dans la listbox sont limitées :



L'utilisation d'un type énumération facilite également la saisie dans la console :

```
enum Drive {  
    CDROM  
    HardDisk  
    Network  
}  
  
Function Test{  
    Param(  
        [Drive] $Drive  
    )  
    'Test'  
}  
  
Test -Drive <TAB>
```

Ce mécanisme est également implémenté pour faciliter la saisie des noms de classes, de méthode ou de propriétés. Dans les exemples précédent Powershell manipule des données statiques.

## 1.2 Complétion dynamique

Si l'on souhaite que les valeurs proposées proviennent d'une source de données quelconque et non plus extraient du code source ou du code compilé d'un assembly, le traitement doit être dynamique.

[Cette possible évolution](#) de l'attribut ValidateSet pourrait résoudre ce problème :

```
param(  
    [ValidateSet( { Get-ChildItem -ah | ForEach-Object Name } )]  
    $HiddenFileName  
)
```

Pour le moment nous allons devoir utiliser un objet spécialisé, plus exactement un cmdlet qui en une passe enregistre et déclare dans un scriptblock un traitement de complétion spécialisé.

## 2 Register-ArgumentCompleter

Attention, si vous avez installé le module nommé TabExpansion++, celui-ci déclare une fonction de même nom d'un usage similaire :

```
gcm -noun *completer*
```

CommandType	Name	Version	Source
-----	----	-----	-----
Function	Get-ArgumentCompleter	1.2	TabExpansion++
Function	Register-ArgumentCompleter	1.2	TabExpansion++
Function	Update-ArgumentCompleter	1.2	TabExpansion++
Cmdlet	Register-ArgumentCompleter	3.0.0.0	Microsoft.PowerShell.Core

Il existe deux approches pour gérer la complétion d'argument, celle basée sur le module TabExpansion++ fonctionnant à partir de la version 3 de de Powershell ou nativement à partir de la version 5. Ce document ne couvre que cette dernière possibilité.

Pour déclarer un traitement de complétion d'argument personnalisé on doit préciser le nom de la commande, le nom du paramètre concerné ainsi qu'un bloc de script.

Note : Les paramètres dynamiques ne nécessitent pas de traitement particulier.

Le scriptblock doit déclarer les paramètres suivants :

```
param(  
    $commandName, $parameterName, $wordToComplete,  
    $commandAst,$fakeBoundParameter  
)
```

Ce sont ceux de la méthode *CompleteArgument* de l'interface **IArgumentCompleter** :

```
IEnumerable<CompletionResult> (  
    string commandName,  
    string parameterName,  
    string wordToComplete,  
    CommandAst commandAst,  
    IDictionary fakeBoundParameters  
)
```

Description des paramètres :

<i>commandName</i>	Le nom de la commande concernée par la complétion d'argument.
<i>parameterName</i>	Le nom du paramètre concerné par la complétion d'argument.
<i>wordToComplete</i>	Le mot en cours de saisie qui est à compléter (il peut être vide).
<i>commandAst</i>	L'AST de la commande en cours de saisie
<i>fakeBoundParameters</i>	Ce paramètre est similaire à <i>\$PSBoundParameters</i> , bien que parfois PowerShell ne peut pas ou ne tente pas d'évaluer un argument, auquel cas vous devrez utiliser <i>commandAst</i> .

**Attention toute erreur dans le code n'est pas signalée et rend la déclaration inopérante.**

Si le module PSReadline n'est pas chargé en mémoire, le comportement par défaut de la touche TAB est de proposer pour une valeur d'argument les fichiers présents dans le répertoire courant :

```
PS C:\temp> get-command -verb .\test.txt
```

Avec l'exemple suivant :

```
Register-ArgumentCompleter -CommandName Get-Command -ParameterName Verb -
ScriptBlock {
    param($commandName, $parameterName, $wordToComplete, $commandAst,
    $fakeBoundParameter)

    write-warning "`r`n`r`n Date=$(get-date -Format 'h:mm:ss') "
    write-warning "commandName=$commandName"
    write-warning " parameterName=$parameterName"
    write-warning " wordToComplete=$wordToComplete"
    write-warning " commandAst=$commandAst"
    write-warning "fakeBoundParameter=$(($fakeBoundParameter|Out-String))"

    Get-Verb "$wordToComplete*" |
        ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_.Verb,
            $_.Verb, 'ParameterValue', ("Group: " + $_.Group))
        }
}
```

La saisie de la touche TAB affiche la liste des verbes renvoyés par le cmdlet **Get-Verb** :

```
PS C:\temp> get-command -verb AddNING:
Date =10:51:17
WARNING: commandName =Get-Command
WARNING: parameterName =Verb
WARNING: wordToComplete =
WARNING: commandAst =get-command -verb
WARNING: fakeBoundParameter =
```

Le paramètre *wordToComplete* est vide car nous n'avons pas saisi de lettre, mais uniquement la touche TAB. Dans ce cas PS ne complète pas une saisie mais propose la première valeur de la liste.

Avec PSReadline la saisie de *control-espace* affiche soit la liste de tous les verbes, soit ceux débutant par les caractères insérés :

```
[STA] C:\temp> get-command -Verb Add_
Add Lock Resize Watch E
Clear Move Search Backup G
Close New Select Checkpoint I
Copy Open Set Commande I
```

## Note :

Une fois saisie la ligne suivante :

```
Get-Command -verb L
```

La première saisie de la touche TAB déclenche l'appel du scriptblock, les saisies suivantes parcourent uniquement la liste des valeurs renvoyées par le scriptblock.

Powershell reconstruit la liste de choix de valeur à partir du moment où le paramètre *\$wordToComplete* est de nouveau renseigné que ce soit en ajoutant ou en supprimant un caractère à la valeur en cours de saisie.

### 2.1 Détail du type résultat : CompletionResult

Le scriptblock associé au cmdlet **Register-ArgumentCompleter** doit retourner une collection d'objet de type **CompletionResults** :

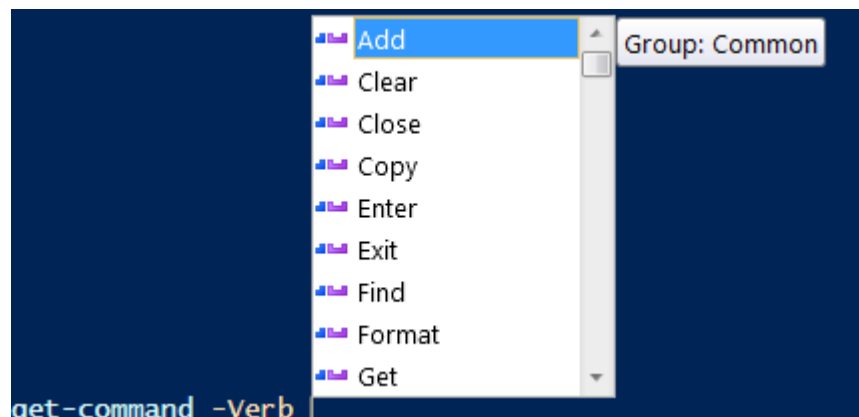
```
public CompletionResult(  
    string completionText,  
    string listItemText,  
    CompletionResultType resultType,  
    string toolTip  
)
```

Description des paramètres :

completionText	Le texte à utiliser comme résultat la complétion.
listItemText	Le texte à afficher dans une liste.
resultType	Valeur de l'énumération <i>CompletionResultType</i> qui représente le type de résultat de la complétion.
toolTip	Le texte de l'info-bulle détaillant l'objet ( <i>completionText</i> ).

La plupart du temps la valeur de la propriété *ResultType* est positionnée à '**ParameterValue**'.

Dans notre exemple l'info-bulle contient le nom du groupe auquel appartient le verbe, l'éditeur ISE l'affiche ainsi :



## 2.2 Déclaration unique pour de multiple commandes

Si on respecte la cohérence de nommage des cmdlets et de leurs paramètres, la déclaration peut porter sur plusieurs commandes :

```
[string[]] $CommandNames=(Get-Command -Noun PSRepository).Name|Sort-Object|Get-Unique

$PSRepositories=Get-PSRepository|Select Name,SourceLocation

$sbRepositoryNames= {
    param($commandName, $parameterName, $wordToComplete, $commandAst,
    $fakeBoundParameter)
    $PSRepositories|
        where{$_.Name -like "$wordToComplete*"}|
        Foreach-{[System.Management.Automation.CompletionResult]::new($_.Name,
    $_.Name, 'ParameterValue', $_.SourceLocation) }
}

Register-ArgumentCompleter -CommandName $CommandNames -ParameterName Name
-ScriptBlock $sbRepositoryNames

$CommandNames =(Get-Command -ParameterName Repository).Name
Register-ArgumentCompleter -CommandName $CommandNames -ParameterName
Repository -ScriptBlock $sbRepositoryNames
```

Afin d'éviter un délai d'attente lors de la complétion, la construction de la liste des noms de repository Powershell ne se fait pas dans le code du scriptblock. La mise en cache de cette liste est une évolution souhaitable et de prêter une attention au scope ne sera pas superflue ☺.

Le module TabExpansion++ propose par défaut une gestion de cache couvrant ce cas.



### 3 L'attribut [ArgumentCompleter()]

Nous venons de voir l'ajout de la complétion sur une commande existante, il existe une autre possibilité de déclarer une complétion d'argument dès la conception en déclarant sur un paramètre l'attribut `[ArgumentCompleter()]`.

On peut le paramétrer soit avec un scriptblock soit avec un type.

#### 3.1 Paramétrer avec un scriptblock

La complétion suivante construit la liste des noms d'assemblies chargées en mémoire :

```
Function Get-ExportedTypes {
Param (
    [ArgumentCompleter( {
        [AppDomain]::CurrentDomain.GetAssemblies() |
        where-Object {$null -ne $_.Location } |
        Foreach-Object {Split-path $_.Location -Leaf } |
        Sort |
        Get-Unique
    })]
    [string] $Assembly
)
}
```

Vous remarquerez qu'ici le résultat n'est pas renvoyé dans des objets de type *CompletionResult* mais en tant que chaîne de caractères :

```
[STA] C:\temp> Get-ExportedTypes -Assembly ILSpy.exe
ILSpy.exe          System.Configuration.Install.dll
Microsoft.CSharp.dll System.Core.dll
Microsoft.Management.Infrastructure.dll System.Data.dll
Microsoft.PowerShell.Commands.Management.dll System.DirectoryServices.dll
```

En interne le résultat est transformé en type *CompletionResult* à partir d'un tableau de chaînes, ce qui facilite l'écriture du code du scriptblock.

Le code suivant

```
Function Get-ExportedTypes {
Param (
    [ArgumentCompleter( { [AppDomain]::CurrentDomain.GetAssemblies() })]
    [string] $Assembly
)
}
```

Renvoie ceci :

```
[ISTA] C:\temp> Get-ExportedTypes -Assembly mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
Microsoft.PowerShell.ConsoleHost, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
```

Qui est similaire à :

```
$A=[AppDomain]::CurrentDomain.GetAssemblies(); $A[0].ToString()
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

### 3.2 Paramétrer avec un type

On peut également lui passer en paramètre un type. Celui-ci doit obligatoirement implémenter l'interface **IArgumentCompleter** que nous avons abordé précédemment.

Cette approche cible à l'origine le code C#, mais depuis la version 5 il est possible de l'implémenter à l'aide des classe Powershell natives :

```
#Extrait de : #PowerShell/test/powershell/Language/Parser/ExtensibleCompletion.Tests.ps1
using namespace System.Management.Automation
using namespace System.Management.Automation.Language
using namespace System.Collections
using namespace System.Collections.Generic
CLASS VerbArgumentCompleter : IArgumentCompleter
{
    [IEnumerable[CompletionResult]] CompleteArgument(
        [string] $CommandName,
        [string] $parameterName,
        [string] $wordToComplete,
        [CommandAst] $commandAst,
        [IDictionary] $fakeBoundParameters )
    {
        $resultList = [List[CompletionResult]]::new()
        Get-Verb "$wordToComplete*" |
        ForEach-Object {
            $resultList.Add([CompletionResult]::new($_.Verb, $_.Verb,
'ParameterValue', ("Group: " + $_.Group)))
        }
        return $resultList
    }
}
Function Test {
    Param (
        [ArgumentCompleter([VerbArgumentCompleter])]
        [string] $Verb)
}
```

Concis et pratique !

### 3.3 Combinaison d'attributs

La déclaration conjointe de l'attribut *ValidateSet* supprime celle de *ArgumentCompleter* :

```
Function Test {  
    Param (  
        [ArgumentCompleter([VerbArgumentCompleter])]  
        [ValidateSet('Debug', 'Measure', 'Ping', 'Repair', 'Resolve', 'Test', 'Trace')]  
        [string] $Verb  
    )  
}
```

Et celle d'un paramètre de type énumération avec un *ArgumentCompleter* également :

```
enum Diagnostic {  
    Debug  
    Measure  
    Ping  
    Repair  
    Resolve  
    Test  
    Trace  
}  
  
Function Test {  
    Param (  
        [ArgumentCompleter([VerbArgumentCompleter])]  
        [Diagnostic] $Verb  
    )  
}
```

La combinaison d'un paramètre de type énumération avec l'attribut *ValidateSet* reste possible.

### 3.4 Masquer un paramètre

L'attribut *[Parameter()]* propose depuis la version 4 une propriété nommée ***DontShow*** de type booléen. Elle exclut le paramètre de la liste des paramètres gérés par la touche <Tab>.

Ceci est principalement utilisé dans les fonctions implémentant des paramètres dynamiques :

```
DynamicParam {  
    $MyAttribute = New-Object System.Management.Automation.ParameterAttribute  
    $MyAttribute.DontShow = $true  
    ...  
}
```

## 4 Complétion de commandes natives

Les programmes externes, tel que Powershell.exe, n'utilisent pas la gestion de paramètre spécifique à Powershell, il est nécessaire de créer la liste de ses paramètres afin de les proposer lors de la complétion.

**Register-ArgumentCompleter** propose le switch *-Native* qui en interne provoque l'insertion du compléteur dans une liste dédiée d'accès privé.

Je reprends ici l'exemple de la documentation :

```
Register-ArgumentCompleter -Native -CommandName powershell -ScriptBlock {
    param($wordToComplete, $commandAst, $cursorPosition)

    echo -- -PSConsoleFile -Version -NoLogo -NoExit -Sta -NoProfile -
NonInteractive
        -InputFormat -OutputFormat -WindowStyle -EncodedCommand -File -
ExecutionPolicy
        -Command |
        where-Object { $_ -like "$wordToComplete*" } |
        Sort-Object |
        ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_, $_,
'ParameterValue', $_)
        }
}
```

Dans ce cas ces noms de paramètre sont bien vus comme des arguments de la commande native. Mais on ne peut pas saisir *-File* puis <TAB> pour sélectionner un nom de fichier, ça ne fonctionne pas.

Vous trouverez sur le site de TabExpansionPlusPlus de nombreux exemples, [on constate](#) que ce module simplifie la création de compléteur de commande native.

## 5 Inférence de type

Définition de l'inférence : *Opération qui consiste à admettre une proposition en raison de son lien avec une proposition préalable tenue pour vraie.*

Dans le contexte d'un langage informatique, il s'agit de déduire selon le contexte le type d'une variable qui n'est pas précisé.

Dans l'exemple suivant et grâce à l'inférence de type, le type *Diagnostics.Process* est déterminé à partir du type de l'objet émis\* précédemment dans le pipeline :

```
Get-Process | Foreach-Object {$_.<tab>
```

La complétion peut donc proposer la liste des membres du type concerné.

\* Pour cet exemple, Powershell n'exécute pas le code mais utilise une analyse statique de l'AST.

La version 6 proposera cette possibilité qui ne fonctionne pas à ce jour :

```
[Diagnostics.Process[]] $p = Get-Process  
$p | % Prio<tab>
```

Dans le premier cas le cmdlet aide le mécanisme de complétion en déclarant l'attribut *OutputType*.

### 5.1 L'attribut *OutputType*

La complétion ne recherche pas d'informations sur la classe de **Get-Process** mais sur le type de l'objet qu'il retourne, ici le type *Diagnostics.Process*.

Le rôle de l'attribut *OutputType* consiste à préciser le type de la valeur de retour associée au jeu de paramètre actif. Sachez que Powershell ne contrôle pas la cohérence entre le type précisé et le type réellement renvoyé par la fonction.

En revanche pour les fonctions avancées, la présence de plusieurs déclarations de cet attribut, par exemple un par jeu de paramètre, ne semble pas prise en compte.

### 5.2 Contexte de complétion dans *PSReadLine*

Pour l'exemple suivant la combinaison *Control-Espace* ne propose aucun choix :

```
rv Var  
$var.
```

Ce qui est normal puisque la variable **\$Var** n'existe pas.

Si on déclare notre variable sur la même ligne :

```
[ISTA] C:\temp> $Var=10; $Var.CompareTo(  
CompareTo GetType ToBoolean ToI
```

La combinaison *Control-Espace* propose la liste des membres du type affecté à **\$Var**.

Si on annule la saisie par la touche 'Échap', on constate que notre variable n'existe pas :

```
Get-Variable Var  
Get-variable : Cannot find a variable with the name 'Var'.
```

Le contexte dans ce cas est la ligne en cours de saisie, mais ce comportement est celui du module *PSReadline*. Si on le supprime de la session, cette fonctionnalité n'existera plus.

### 5.3 fakeBoundParameter

On peut utiliser ce paramètre du compléteur pour proposer une liste de valeur selon le contenu d'un paramètre précédent.

Prenons le cas du cmdlet *Group-Object* son paramètre *-InputObject* attend une collection d'objet et *-Property* attend le nom de la propriété de regroupement :

```
$sb= {  
    param($commandName, $parameterName, $wordToComplete, $commandAst,  
    $fakeBoundParameter)  
    $fakeBoundParameter |out-string|Foreach-Object {write-host $_}  
}  
Register-ArgumentCompleter -CommandName Group-Object -ParameterName  
property -ScriptBlock $sb
```

Pour le cas suivant :

```
Group-Object -InputObject (Get-Process) -Property <TAB>
```

Le paramètre *\$fakeBoundParameter* n'est pas renseigné. Mais il l'est avec celui-ci :

```
$P=Get-Process  
Group-Object -InputObject $F -Property <TAB>
```

Ainsi on peut récupérer les noms des propriétés du type de l'argument à partir de la hashtable :

```
$sb= {  
    param($commandName, $parameterName, $wordToComplete, $commandAst,  
    $fakeBoundParameter)  
    $fakeBoundParameter.InputObject[0].GetType().GetMembers() |  
    where {$_.MemberType -eq 'Property'} |select -expandproperty Name |  
    where{$_ -like "$wordToComplete*"} |  
    Foreach{  
        [System.Management.Automation.CompletionResult]::new(  
            $_, $_, 'ParameterValue', $_)  
        }  
    }  
}  
Register-ArgumentCompleter -CommandName Group-Object -ParameterName  
property -ScriptBlock $sb  
  
$f=dir $Env:windir
```

```
[STA] C:\temp> Group-Object -InputObject $F -Property Name  
Name  
Exists  
CreationTime  
LastAccessTimeUtc  
Attributes  
FullName  
Root  
CreationTimeUtc  
LastWriteTime  
Parent  
Extension  
LastAccessTime  
LastWriteTimeUtc
```

Le principe fonctionne, notez que le code nécessiterait d'autres contrôles. Cet exemple n'est pas très utile, puisque dans ce cas le cmdlet ne crée qu'un seul groupe.

Pour récupérer plusieurs groupes l'exemple suivant ne renseigne pas le paramètre *\$fakeBoundParameter* :

```
$p|Group-Object -Property <TAB>
```

Dans ce cas on doit utiliser l'AST précisé dans le paramètre *\$CommandAst*, plus précisément celle de son parent :

```
$sb= {  
    param($commandName, $parameterName, $wordToComplete, $commandAst,  
    $fakeBoundParameter)  
    $vn=$commandAst.Parent.PipelineElements[0].Expression.VariablePath.UserPath  
    $type=(Get-variable $vn).GetType()  
    $type.GetMembers() |  
        where {$_.MemberType -eq 'Property'} | select -expandproperty Name |  
        where {$_ -like "$wordToComplete*"} |  
        Foreach{  
            [System.Management.Automation.CompletionResult]::new(  
                $_, $_, 'ParameterValue', $type)  
        }  
}  
  
Register-ArgumentCompleter -CommandName Group-Object -ParameterName  
property -ScriptBlock $sb
```

La construction d'une liste de valeur pour compléter un paramètre est relativement simple, par contre l'analyse des différentes syntaxes et possibilités nécessitera plus d'efforts.

## 6 Les fonctions de hook

Ces fonctions sont prévues pour personnaliser certains traitements, à la différence [des hook système](#) il n'est pas nécessaire d'intercepter des appels de fonction ou de déclarer des gestionnaires d'événements. Une déclaration ou une redéfinition de fonction suffit, Powershell se charge du reste.

### 6.1 TabExpansion

Powershell propose les fonctions *TabExpansion* et *TabExpansion2* en tant que point d'entrée vers la complétion. Elle permet de configurer la complétion des propriétés, paramètres et fichiers.

La fonction nommée *TabExpansion* est utilisée par la version 2 de Powershell et celle nommée *TabExpansion2* à partir de la version 3. La principale différence étant que la dernière accède à l'AST là où la première utilisait principalement des expressions régulières.

Pour afficher son contenu :

```
$(function:TabExpansion2)
```

Powershell crée cette fonction et l'exécute lors de l'appel de la complétion, à son tour elle appelle la [méthode statique](#) suivante :

```
[System.Management.Automation.CommandCompletion]::CompleteInput(...
```

La complétion cesse de fonctionner si on affecte *\$null* à cette fonction:

```
$(function:TabExpansion2)=$null
```

### 6.2 Lecture de la saisie utilisateur

Powershell version 3 propose également la possibilité de redéfinir la lecture clavier à l'aide de la fonction *PSConsoleHostReadLine*.

Le module PSReadline, le bien nommé, la redéfinit ainsi :

```
function PSConsoleHostReadline
{
    Microsoft.PowerShell.Core\Set-StrictMode -Off
    [Microsoft.PowerShell.PSConsoleReadline]::ReadLine($host.$Runspace,
    $ExecutionContext)
}
```

Son rôle est d'intercepter toute saisie utilisateur afin de déclencher des traitements. Par exemple ce module associe par défaut la touche TAB à une de ses fonctions C# internes :

```
Get-PSReadlineKeyHandler -Bound|where {$_.Key -eq 'Tab'}
```

Key	Function	Description
---	-----	-----
Tab	TabCompleteNext	Complete the input using the next completion

Vous trouverez des exemples de configuration personnalisée dans le fichier suivant :

```
"$((Get-Module PSReadline).ModuleBase\SamplePSReadlineProfile.ps1"
```



### 6.3 Résolution de commande

Depuis la version 3 Powershell permet d'intervenir sur le traitement d'une commande, une fois celle-ci complétée, à l'aide de trois propriétés :

```
$ExecutionContext.SessionState.InvokeCommand|  
gm -MemberType Property|  
select name | fl
```

```
Name : CommandNotFoundAction  
Name : PostCommandLookupAction  
Name : PreCommandLookupAction
```

Celles-ci contiennent le code d'un [gestionnaire d'événement](#) qui reçoit en [paramètre](#) le nom de la commande et les données reçues par le gestionnaire d'événement.

L'exemple suivant gère une commande inexistante nommée '**DirC**' :

```
$ExecutionContext.SessionState.InvokeCommand.PreCommandLookupAction = {  
    param($CommandName, $CommandLookupEventArgs)  
    #Event handler that is called before the command lookup  
    if ($CommandName -eq 'DirC')  
    {  
        write-warning "PreCommandLookupAction : $CommandName"  
        #write-host ($CommandLookupEventArgs|select *|out-string)  
    }  
}  
  
$ExecutionContext.SessionState.InvokeCommand.CommandNotFoundAction = {  
    param($CommandName, $CommandLookupEventArgs)  
    # Event handler that is called when a command is not found.  
    write-warning "CommandNotFoundAction : $CommandName"  
}  
  
$ExecutionContext.SessionState.InvokeCommand.PostCommandLookupAction = {  
    param($CommandName, $CommandLookupEventArgs)  
    #Event handler that is called after the command lookup is done  
    # but before the event object is returned to the caller,  
    # to allow other events, such as interning scripts, to work correctly.  
    if ($CommandName -eq 'DirC')  
    { write-warning "PostCommandLookupAction : $CommandName" }  
}
```

Une fois les propriétés affectées, l'exécution affiche:

```
DirC
WARNING: PreCommandLookupAction : dirc
WARNING: CommandNotFoundAction : get-dirc
WARNING: CommandNotFoundAction : dirc
dirc : The term 'dirc' is not recognized as the name of a cmdlet,
function, script file, or operable program. ...
WARNING: CommandNotFoundAction : .\dirc
```

La commande n'étant pas trouvée le code de *PostCommandLookupAction* n'est pas exécuté. Ceci est dû au fait que le code de *CommandNotFoundAction* n'intervient pas sur la résolution de commande.

Maintenant modifions le en remplaçant la commande reçue par une nouvelle commande :

```
$ExecutionContext.SessionState.InvokeCommand.PreCommandLookupAction = {
    param($CommandName, $CommandLookupEventArgs)
    if ($CommandName -eq 'DirC')
    {
        write-warning "PreCommandLookupAction : $CommandName"
        #Arrête l'exécution des event handler
        #$CommandLookupEventArgs.StopSearch = $true
    }
    if ($CommandName -eq 'Dire')
    {
        write-warning "PreCommandLookupAction : $CommandName"
        #Remplace une commande par une autre (existante ou pas)
        $CommandLookupEventArgs.CommandScriptBlock={ DirC }
    }
}

$ExecutionContext.SessionState.InvokeCommand.CommandNotFoundAction = {
    param($CommandName, $CommandLookupEventArgs)
    write-warning "CommandNotFoundAction : $CommandName"
    if ($CommandName -eq 'DirC')
    {
        #Déclenche PostCommandLookupAction puis exécute ce code
        $CommandLookupEventArgs.CommandScriptBlock={ Dir C:\}
    }
}
```

```
$ExecutionContext.SessionState.InvokeCommand.PostCommandLookupAction = {
    param($CommandName, $CommandLookupEventArgs)
    if ($CommandName -eq 'DirC')
    { Write-Warning "PostCommandLookupAction : $CommandName" }
}

#Trace-Command CommandDiscovery { DirC } -PSHost
```

Dans le code de **CommandNotFoundAction** on affecte une nouvelle commande à l'aide de la propriété *\$CommandLookupEventArgs.CommandScriptBlock*. En interne cette affectation attribut la valeur \$true à la propriété *\$CommandLookupEventArgs.StopSearch*.

Pour le texte 'DirC, qui est considéré comme un nom de commande,' Powershell exécute le code du scriptblock { 'Dir C:\' } :

```
DirC
WARNING: PreCommandLookupAction : dirc
WARNING: CommandNotFoundAction : get-dirc
WARNING: CommandNotFoundAction : dirc
WARNING: PostCommandLookupAction : dirc
    Directory: C:\

Mode                LastWriteTime         Length Name
----                -
...

```

Notez que si le code de *PreCommandLookupAction* reçoit le texte 'Dire' il émet le résultat d'exécution de la propriété *CommandScriptBlock*, c'est à dire 'DirC'. C'est donc une nouvelle commande qui déclenche à nouveau l'événement *PreCommandLookupAction* :

```
Dire
WARNING: PreCommandLookupAction : dire
WARNING: PreCommandLookupAction : DirC
WARNING: CommandNotFoundAction : get-DirC
WARNING: CommandNotFoundAction : DirC
WARNING: PostCommandLookupAction : DirC
    Directory: C:\ ...

```

Ces gestionnaires d'événement sont donc exécutés pour chaque nom de commande à résoudre.

Si on crée une fonction nommée *DirC*, en laissant en l'état les gestionnaires actuels, seul l'événement **CommandNotFoundAction** n'est plus exécuté :

```
function DirC{'Test'}  
DirC  
WARNING: PreCommandLookupAction : DirC  
WARNING: PostCommandLookupAction : DirC  
Test
```

*Notes :*

- Dans le code du gestionnaire on ne récupère que le nom de commande et pas l'intégralité de la ligne saisie.
- L'historique ne considère que la ligne saisie et pas la nouvelle commande.
- Ces gestionnaires s'appliquent au code saisi dans la console et au code exécuté via un script, un module ou un cmdlet.