

# Analyse de code Powershell, Token et AST

Par Laurent Dardenne, le 22/06/2014.



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

Conçu avec Powershell v3 sous Windows Seven 64 bits.

Merci à Matthew Betton pour sa relecture.

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

Les fichiers sources :

<http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/AnalyseDeCodePowershell/Sources.zip>

# Chapitres

<b>1</b>	<b>PREAMBULE</b> .....	<b>3</b>
<b>2</b>	<b>RAPPEL DE GRAMMAIRE</b> .....	<b>3</b>
<b>3</b>	<b>UN SOUPÇON D'ANALYSE LEXICALE</b> .....	<b>5</b>
3.1	RETROUVER UN TYPE DE TOKEN .....	7
3.2	RECHERCHE DE DEPENDANCES.....	9
3.3	TRANSFORMATION DE CODE .....	10
3.3.1	<i>Manipulation du texte</i> .....	11
3.4	LIMITE D'UNE LISTE DE TOKENS.....	12
<b>4</b>	<b>UN PEU D'ANALYSE SYNTAXIQUE</b> .....	<b>13</b>
4.1	ABSTRACT SYNTAX TREE (AST) .....	14
4.2	STRUCTURE D'UN NŒUD .....	14
4.3	AUPRES DE MON ARBRE .....	17
4.4	TROUVEZ-LES TOUS !.....	19
4.5	UN COMMENTAIRE ? .....	20
<b>5</b>	<b>UNE VISITE GUIDÉE</b> .....	<b>21</b>
5.1	QU'EST-CE QUE CETTE VISITE ?.....	21
5.2	VISITEUR C# .....	22
5.3	VISITEUR BASE MODULE .....	23
5.4	VISITEUR BASE SCRIPTBLOCK.....	24
5.5	LIMITES.....	25
<b>6</b>	<b>SCRIPT ANALYZER</b> .....	<b>26</b>
<b>7</b>	<b>EXEMPLE : ANALYSE D'UNE LIGNE DE COMMANDE</b> .....	<b>26</b>
7.1	PRINCIPE .....	27
7.2	CONTRAINTES .....	27
7.3	LIMITES.....	28
<b>8</b>	<b>EXEMPLE : REECRITURE D'UNE INSTRUCTION</b> .....	<b>29</b>
8.1	PRINCIPE .....	29
8.2	LIMITES.....	30
<b>9</b>	<b>CONCLUSION</b> .....	<b>30</b>

## 1 Préambule

Depuis la version 2 les API de Powershell proposent un moyen d'analyser du code Powershell. Au travers de cette analyse il est possible d'effectuer des recherches sur des éléments du langage, par exemple connaître le nom de toutes les variables ou fonctions déclarées dans un script, de transformer du code ou encore de contrôler certains points.

Elles permettent d'automatiser non pas des tâches d'infrastructures, mais des traitements sur le code source. Ce tutoriel s'adresse en premier lieu à des développeurs, plus précisément aux outilleurs *alias* 'tools maker'.

## 2 Rappel de grammaire

Avant d'aborder le sujet voyons quelques éléments de base.

Un code source est constitué de mot clés d'un langage informatique respectant une syntaxe, dans l'exemple suivant :

```
$Var=10
If ($var -gt 8) {Write-Host 'Le contenu de $Var est supérieure à 8'}
Else {Write-Host 'Le contenu de $Var est inférieure à 8'}
```

**\$Var** est une variable, **10** et **8** des constantes, **If** et **else** sont des mots clés du langage, **-eq** un opérateur et **Write-Host** un nom de commande, tout comme un script, une fonction, ou un programme externe.

Les règles d'assemblage de ces mots clés sont déterminées par une grammaire pouvant être représentée dans un formalisme particulier, ci-dessous la règle du test conditionnel IF :

```
<ifStatementRule> =
'if' '(' <pipelineRule> ')' <statementBlockRule> [
'elseif' '(' <pipelineRule> ')' <statementBlockRule> ]*
[ 'else' <statementBlockRule> ]{0|1}
```

On y trouve le nom de la règle *ifStatementRule*, les mots clés **If**, **else** et **elseif**, ainsi que des renvois vers d'autres règles. Il est également précisé que la valeur à tester doit être précédée d'une parenthèse ouvrante et suivi d'une parenthèse fermante.

Pour cette construction l'instruction **If** est unique et obligatoire, l'instruction **elseif** peut être multiple (\*) ou absente, l'instruction **else** est optionnelle et dans ce cas unique (0/1).

Cette règle provient du document nommé 'GrammairePowershell.pdf', présent dans le répertoire \Documentation de l'archive des sources d'exemple.

Ce document détaille la grammaire de Powershell à l'aide de la [forme de Backus-Naur](#) (BNF).

La spécification de cette règle se trouve dans le document nommé '*PowerShell 3.0 Language Specification.docx*' :

### **The if statement**

#### **Syntax:**

#### **Description:**

The *pipeline* controlling expressions must have type `bool` or be implicitly convertible to that type. The *else-clause* is optional. There may be zero or more *elseif-clauses*.

If the top-level *pipeline* tests `True`, then its *statement-block* is executed and execution of the statement terminates. Otherwise, if an *elseif-clause* is present, if its *pipeline* tests `True`, then its *statement-block* is executed and execution of the statement terminates. Otherwise, if an *else-clause* is present, its *statement-block* is executed.

### 3 Un soupçon d'analyse lexicale

Ces règles permettent d'implémenter un traitement de récupération des éléments d'un code source. Lors du parcourt du texte il récupère, tout mot, groupe de mots ou de caractères qui, dans ce contexte, sont appelés "token" (unités lexicales ou lexèmes).

Voici un exemple de code source :

```
$Code=@'  
function maFonction($Serveur)  
{  
    write-host "Dans MaFonction"  
}  
'@
```

et l'appel de l'API d'analyse lexicale à l'aide la méthode statique *Tokenize()* :

```
[ref]$Errors = $null  
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $Errors)  
$Result.Count  
12  
$Errors.Value.Count  
0
```

La variable **\$Result** contient le code source transformé en une collection de *tokens*, ici la variable **\$Errors** est une collection vide.

Le texte du code source est constitué de 12 *tokens* catégorisés :

```
IMTA] C:\Temp> $Result|Format-Table -AutoSize
```

Content	Type	Start	Length	StartLine	StartColumn	EndLine	EndColumn
function	Keyword	2	8	1	3	1	11
maFonction	CommandArgument	11	10	1	12	1	22
<	GroupStart	21	1	1	22	1	23
Serveur	Variable	22	8	1	23	1	31
>	GroupEnd	30	1	1	31	1	32
...	NewLine	31	1	1	32	1	33
<	GroupStart	34	1	2	3	2	4
...	NewLine	35	1	2	4	2	5
Write-host	Command	40	10	3	5	3	15
Dans MaFonction	String	51	17	3	16	3	33
...	NewLine	68	1	3	33	4	1
>	GroupEnd	71	1	4	3	4	4

#### Note :

Attention le token '.' à deux usages, c'est un opérateur d'accès à un membre et un opérateur d'invocation de code dans la portée courante.

L'[analyse lexicale](#) se fait sur l'intégralité d'un code source, celle-ci renvoi une collection d'objets de type **PSToken** :

```
$Result[0] | Get-Member -MemberType property | Select Name, Definition
    TypeName: System.Management.Automation.PSToken
Name          Definition
----          -
Content       System.String Content {get;}
EndColumn     System.Int32 EndColumn {get;}
...
StartLine     System.Int32 StartLine {get;}
Type          System.Management.Automation.PSTokenType Type {get;}
```

Toutes ses propriétés précisent la position d'un token dans le texte du code source, sauf la propriété [Type](#) qui indique si le token est une commande, ou un opérateur, etc.

Ces propriétés peuvent, par exemple, être couplées dans ISE avec les méthodes de l'objet `$psISE.CurrentFile.Editor` afin de rechercher/modifier un mot dans le texte de l'éditeur.

Voici l'association texte-token issue de l'analyse de la fonction précédente :

```
$Result | Foreach-Object {
    "Le texte ``{0}`` est un token de type '{1}'" -F $_.Content, $_.Type
}
```

```
Le texte "function" est un token de type 'Keyword'
Le texte "maFonction" est un token de type 'CommandArgument'
Le texte "(" est un token de type 'GroupStart'
Le texte "Serveur" est un token de type 'Variable'
Le texte ")" est un token de type 'GroupEnd'
Le texte "
" est un token de type 'NewLine'
Le texte "{" est un token de type 'GroupStart'
Le texte "
" est un token de type 'NewLine'
Le texte "Write-host" est un token de type 'Command'
Le texte "Dans MaFonction" est un token de type 'String'
Le texte "
" est un token de type 'NewLine'
Le texte "}" est un token de type 'GroupEnd'
```

Cette analyse lexicale 'découpe' notre texte en tokens. Certains caractères tels que '(' et '{' sont dans la même catégorie de token, d'autres tels que les caractères espaces, les tabulations, les caractères '\$', '[', ']' ne sont pas mémorisés.

Notez que l'analyse d'un code incomplet provoquera une erreur :

```
$Code=@'
function maFonction($Serveur)
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $Errors)
```

Pour cette exemple, l'appel à l'API *Tokenize()* renseigne la collection **\$Errors** :

```
#Variable de type référence
$errors.value.Count
1
$errors.value
Token                Message
-----
System.Management.Automation.PSToken Corps de fonction manquant dans la déclaration de fonction.
```

On peut donc dès maintenant mettre à profit cette API dans une fonction de contrôle de syntaxe, comme le permet la fonction '*Test-PSScript.ps1*' disponible dans le répertoire *Source*.

De ce que l'on constate, l'API *Tokenize* effectue une analyse lexicale ET une analyse syntaxique.

On peut penser que ce choix de l'équipe Powershell facilite une partie du travail.

*Note* : Le mot 'token' peut être utilisé dans d'autre contexte et signifier autre chose.

Un exemple : <http://blogs.technet.com/b/askds/archive/2007/11/02/what-s-in-a-token.aspx>

### 3.1 Retrouver un type de token

Une collection de tokens permet également de retrouver tous les noms de fonctions contenus dans un code source :

```
$Code=@'
function Une {Write-host "Fonction Une"}
function Deux {Write-host "Fonction Deux"}
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$result=[System.Management.Automation.PSParser]::Tokenize($Code, $Errors)
$I=0
$result|
where-Object {
    #La variable I pointe sur le token suivant
    #le mot clé Function est toujours suivi d'un identifiant
    $I++
    ($_.Type -eq 'keyword') -and ($_.Content -eq 'function')
}|
Foreach-Object {
    "Fonction $($result[$i].Content)"
}
Fonction Une
Fonction Deux
```

Pour cet exemple une recherche textuelle à l'aide d'expression régulière peut suffire, mais si on souhaite retrouver tous les types de commandes, seule l'analyse d'une liste de tokens peut nous aider :

```
$Code=@'
function Test{
  param(
    [int] $a,
    [Object] $objet=$(Throw "Erreur")
  )
  Dir C:\temp
  Notepad.exe
  Test
  &"C:\temp\Traitement.ps1"
}
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $Errors)
$Result|
  where-Object {
    ($_.Type -eq 'Command') -or (
      ($_.Type -eq 'Operator') -and ($_.Content -match '^(&|\.)$'))
  }|
  Foreach-Object {
    "Commande $_.Content"
  }
```

```
Commande Throw
Commande Dir
Commande Notepad.exe
Commande Test
Commande &
```

Tout compte fait, l'usage de tokens n'exclut pas l'usage de regex.

Ce code d'analyse est incomplet, car il n'affiche pas l'opérande de l'opérateur **&**, en revanche il met en évidence que la réussite de l'analyse syntaxique ne signifie pas un code exempt d'erreur.

En effet, le mot *'Throw'* est reconnu comme un token de type **command** et pas comme le mot clé *'Throw'* mal orthographié. Powershell étant un langage dynamique, ce token sera validé ou invalidé lors de l'exécution du code source associé. C'est un point important à mémoriser.

Pour couvrir ce cas, la construction de tests unitaires est nécessaire. Sachez que la fonction *'Test-PSScript.ps1'*, présentée précédemment, ne valide donc qu'un *'aspect'* du code source testé.



## 3.2 Recherche de dépendances

Nous avons vu que l'on pouvait filtrer la collection de tokens selon leur type, Powershell proposant des métadonnées sur les commandes, nous les utiliserons pour retrouver les dépendances de module. *Ce traitement nécessitera les versions 3 ou supérieures de Powershell*, car elles s'appuient sur le chargement automatique de module :

```
$PSModuleAutoloadingPreference='All'
$RuntimeModules=@(
  'Microsoft.PowerShell.Diagnostics',
  'Microsoft.PowerShell.Host',
  'Microsoft.PowerShell.Management',
  'Microsoft.PowerShell.Security',
  'Microsoft.PowerShell.Utility',
  'Microsoft.WSMan.Management'
)
$Code=@'
function Show-BitsTransfer{
  Write-Host "BitsTransfer report" -fore Green
  Get-BitsTransfer
  Get-Ghost #Erreur
  Disable-PSTrace
}#Show-BitsTransfer
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $Errors)
$Modules=$Result| where-Object {$_.Type -eq 'Command'}|
  Foreach-Object {
    try {
      $CommandName=$_.Content
      $Command=Get-Command $CommandName -EA Stop
      $ModuleName=$Command.ModuleName
      $Version=$Command.Module.Version
      if ($RuntimeModules -NotContains $ModuleName)
      { Write-Output (
"@{ModuleName=""'+$ModuleName+'";ModuleVersion='+$Version+'}' )}
      Write-Host "La commande '$CommandName' dépend du module :"
      Write-Host "`t $ModuleName -> $((Get-Module $ModuleName).ModuleBase)"
    }
    catch {
      Write-Error "Commande inconnue : '$CommandName'"
    }
  }
}
```

```
$OFS=','  
write-host "#Requires -version 3.0"  
write-host "#Requires -Modules $Modules"
```

La commande 'Write-Host' dépend du module :

Microsoft.PowerShell.Utility -> C:\Windows\SysWOW64\WindowsPowerShell\v1.0

La commande 'Get-BitsTransfer' dépend du module :

BitsTransfer -> C:\windows\system32\windowspowershell\v1.0\Modules\BitsTransfer

**Commande inconnue : 'Get-Ghost'**

La commande 'Disable-PSTrace' dépend du module :

PSDiagnostics ->

C:\windows\system32\windowspowershell\v1.0\Modules\PSDiagnostics

La variable \$Module contient la liste des modules nécessaires à l'exécution du code analysé :

```
#Requires -Version 3.0  
#Requires -Modules @{ModuleName="BitsTransfer";ModuleVersion=1.0.0.0},  
                  @{ModuleName="PSDiagnostics";ModuleVersion=1.0.0.0}
```

Cette instruction pourra être insérée dans les premières lignes du script, ainsi, dans le cas où la variable *\$PSModuleAutoloadingPreference* est initialisée à 'None', on est assuré du chargement automatiquement de ces modules. S'il est communément admis que l'on ne peut penser à tout, on peut toutefois se prémunir de certaines erreurs récurrentes.

### 3.3 Transformation de code

La recherche de tokens ou le contrôle de règles sont assez aisées à réaliser, en revanche la modification du code nécessite plus d'attention.

La fonction 'Resolve-Aliases.ps1' substitue chaque nom d'alias par le nom de la commande qu'il référence :

```
cd DemoPath  
$Path='C:\Temp\demo.ps1'  
@'  
cd c:\temp  
dir *.txt |  
gc  
'@ > $Path  
. .\Resolve-Aliases.ps1
```

```
[MTA] C:\Temp> Resolve-Aliases C:\temp\demo.ps1  
Resolved cd to Set-LocationEx  
Resolved dir to Get-ChildItem  
Resolved gc to Get-Content  
  
Set-LocationEx c:\temp  
Get-ChildItem *.txt |  
Get-Content
```

Le code de cette fonction parcourt la liste des tokens et modifie le texte du code source d'origine.

Ici un cas n'a pas été implémenté, un nom de commande peut être dupliqué, ce qui nécessiterais d'ajouter le nom du module/snapins au nom de commande : *ModuleName\CommandName*.

### 3.3.1 Manipulation du texte

Dans les sources de ce tutoriel vous trouverez une fonction effectuant le même traitement, nommée *Expand-Alias*, dédiée à ISE.

ISE propose l'accès au code source chargé dans l'éditeur, au travers de la variable *\$psise.CurrentFile.Editor.text*. On disposera donc d'une collection de tokens et d'une collection de caractères.

On utilisera la première, qui porte les informations de positions, pour modifier la seconde qui contient le code source. L'auteur utilise une astuce pour ne pas invalider la position des autres tokens lors de l'insertion/suppression de texte. Il effectue la modification de texte en partant de la fin, ainsi il n'est pas nécessaire de mémoriser un offset de décalage :

```
[System.Management.Automation.PsParser]::Tokenize($Content, $TokenErrors) |  
  where { $_.Type -eq 'Command' } |  
  Sort StartLine, StartColumn -Desc |  
  ...
```

Le code de l'exemple précédent, *Resolve-Aliases*, héberge la collection de caractères dans une instance d'un *StringBuilder*. Un tel usage permet l'automatisation de certaines tâches de refactoring, là où un éditeur nécessitera une intervention manuelle.

Le module *ExploreAst* contient également une fonction *Expand-Alias* dédiée à la console.

L'archive *PowerGUI-Expand-Alias.zip* contient un addon pour l'éditeur PowerGui, le principe reste identique seul le code de la manipulation de la collection de caractères diffère.

Il est donc préférable de faciliter la réutilisation en découplant les traitements de recherche/transformation de tokens, des traitements de modification du code source.

Pour terminer, une évidence à rappeler :

*Chaque tâche de modification du code source nécessitera de construire la liste des tokens, ou de l'AST, afin d'actualiser les informations de position.*

Voir également :

<https://connect.microsoft.com/PowerShell/feedback/details/804788/escaped-space-character-is-ignored-in-some-condition>

<http://blogs.msdn.com/b/powershell/archive/2013/10/30/using-asts-with-ise-to-make-scripting-more-productive.aspx>

### 3.4 Limite d'une liste de tokens

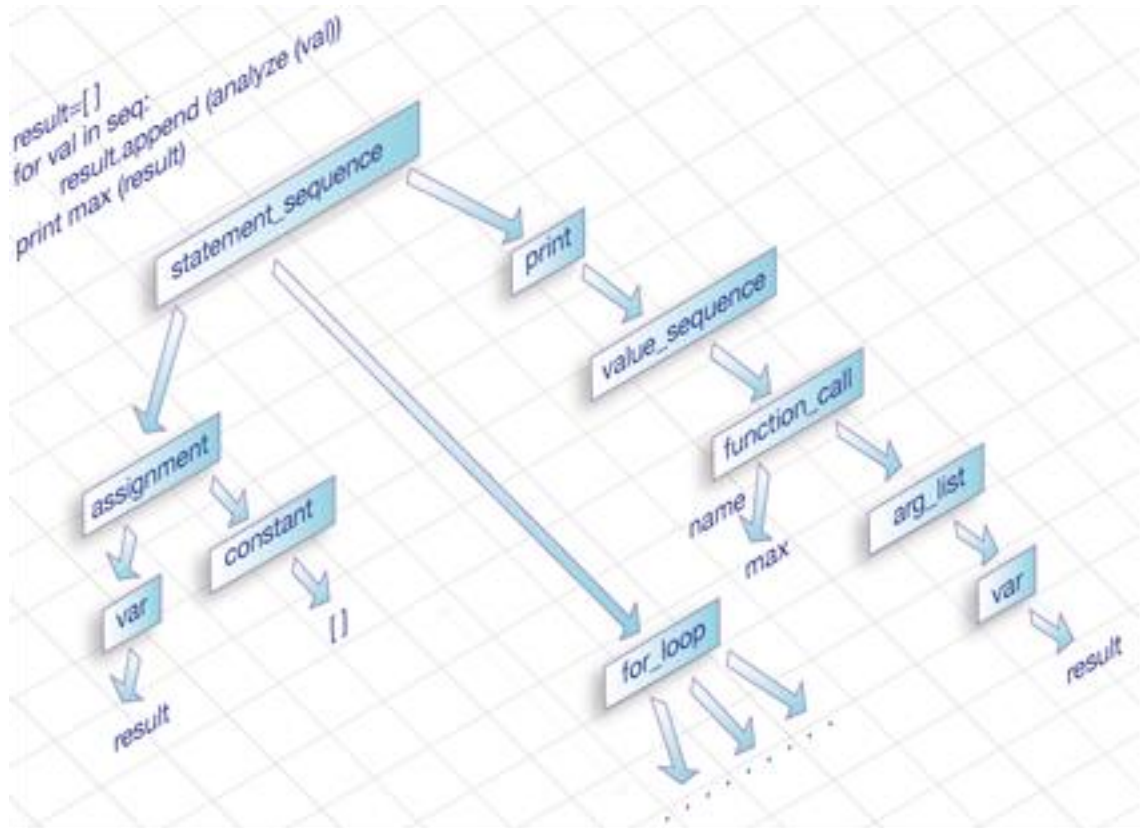
L'usage d'une liste de tokens comme structure de données nécessitera de nombreuses lignes de code et d'efforts pour analyser du code avancé. Prenons par exemple le cas de la recherche des paramètres de cette fonction :

```
$code=@'
function TokenTest{
    param(
        $Name,
        [int] $c, $d, $b = 13,
        [Int32] $Last2=$global:MaxDefaultInt,
        [int][char] $L = "x",
        [Object] $objet=$(Throw "Erreur"),
        $NestedGroup=$(4+$a+($Scopevariable * 2)),
        [System.IO.FileInfo[]] $files = $(dir *.ps1 | sort length),
        $Sb={$_ -eq 10},
        $O2=(Get-myValue 10),
        $Array=@(1,2,3),
        $ht=@{Nom="N";Prenom="P"},
        [ValidateNotNull()]
        [Parameter(Mandatory=$true,valueFromPipeline = $true)]
        [System.Management.Automation.PSObject] $InputObject,
    )
    $a
    $ExpandableString="$ (4+$a+($b * 2))",
}
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $errors)
$Result|Out-GridView
```

On doit recoder les règles de syntaxe du langage Powershell afin de retrouver, dans les multiples manières de définir un paramètre, ceux de cette fonction. De plus le token portant l'information de paramètre n'existe pas ici, il s'agit avant tout de variable.

Notez que la variable *\$ExpandableString* est vue comme une constante de type *string*, son contenu n'est malheureusement pas analysé sous Powershell version 2.

Il manque un accès aux tokens via un arbre de syntaxe, ressemblant au schéma ci-dessous :



<http://woldlab.caltech.edu/~king/swc/www/swc.html>

## 4 Un peu d'analyse syntaxique

Si l'analyse lexicale se préoccupe du découpage des tokens (unités lexicales), l'analyse syntaxique se charge de contrôler leurs règles d'association.

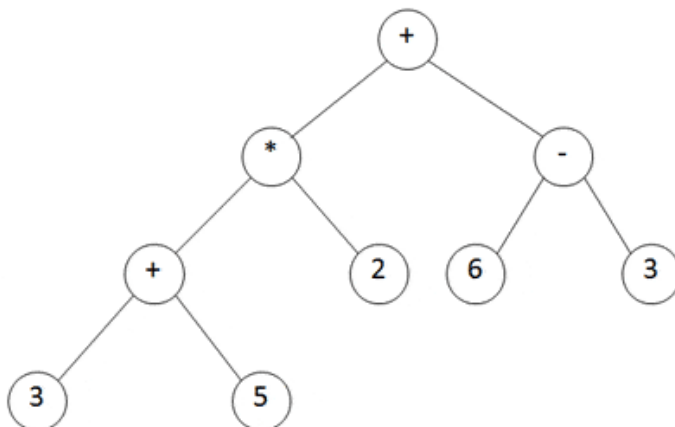
Par exemple l'expression  $(3+5)*2+(6-3)$  est valide, mais pas celle-ci  $3+*2$  :

```
$code=@'
3+*2
'@
[ref]$Errors = [System.Management.Automation.PSParseError[]] @()
$Result=[System.Management.Automation.PSParser]::Tokenize($Code, $errors)
$Result|Out-GridView
$Errors.value|F1
Token : System.Management.Automation.PSToken
Message : Vous devez indiquer une expression de valeur à droite de l'opérateur « + ».
```

La liste des tokens est renseignée tout comme la liste d'erreurs.

## 4.1 Abstract Syntax Tree (AST)

Pour l'expression valide  $(3+5)*2+(6-3)$ , ces éléments sont structurés à l'aide d'un arbre binaire ou arbre de syntaxe abstrait (Abstract Syntax Tree) qui correspondra à ceci :



<http://www.sunshine2k.de/coding/java/SimpleParser/SimpleParser.html>

## 4.2 Structure d'un nœud

Sous Powershell la construction d'un arbre de syntaxe abstrait se fait via la méthode de classe statique `[System.Management.Automation.Language.Parser]::ParseInput`, disponible à partir de Powershell version 3.

Cette méthode peut traiter le contenu d'un fichier ou directement le texte d'un script comme dans l'exemple suivant :

```
$code=' (3+5)*2+(6-3) '
$ErrorsList=$TokensList=$null
$Ast=[System.Management.Automation.Language.Parser]::ParseInput($Code, [ref]
]$TokensList, [ref]$ErrorsList)
```

La variable **\$TokensList** contient tous les tokens, la variable **\$Errorslist** contient les possibles erreurs de syntaxe contenues dans le code source analysé, enfin la variable **\$AST** contient la racine de l'arbre de syntaxe abstrait du code analysé.

Une fois transformée, le code de cette expression est vue comme un objet de type `ScriptBlockAst` :

```
$AST.GetType().FullName
System.Management.Automation.Language.ScriptBlockAst
```

Notez qu'un scriptblock propose une propriété nommée `AST` :

<http://www.powershellmagazine.com/2013/12/23/simplifying-data-manipulation-in-powershell-with-lambda-functions/>

La variable `$AST` contient le nœud racine de l'arbre de syntaxe :

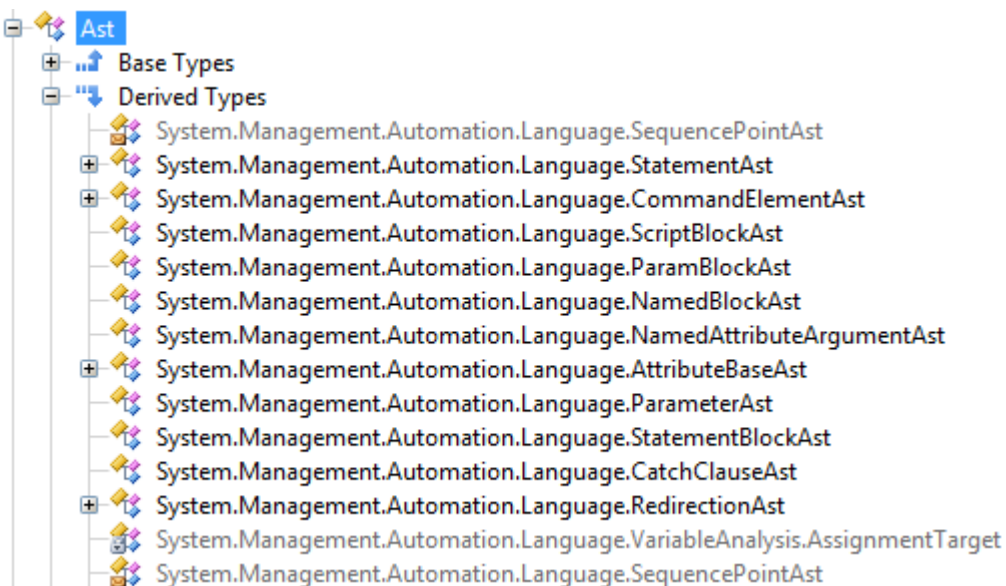
```
$AST
ParamBlock      :
BeginBlock      :
ProcessBlock    :
EndBlock        : (3+5)*2+(6-3)
DynamicParamBlock :
ScriptRequirements :
Extent          : (3+5)*2+(6-3)
Parent          :
```

Comme c'est un arbre on ne connaît plus directement le nombre d'élément contenu.

Tous les types de nœuds de cet arbre dérivent de la classe `Ast` :

```
$AST.psobject.TypeNames
System.Management.Automation.Language.ScriptBlockAst
System.Management.Automation.Language.Ast
System.Object
```

Dans la recopie d'écran suivant les classes grisées sont internes au runtime Powershell :



Certains tokens nécessitent que leur classe AST associée porte plus d'informations que d'autres.

L'analyse du code de cette expression ne renseigne que la propriété `EndBlock`, pour une fonction avancée les propriétés `ParamBlock`, `BeginBlock`, `ProcessBlock`, et `EndBlock` pourront être renseignées, sous réserve de spécifier le code de la fonction :

```
[System.Management.Automation.Language.Parser]::ParseInput($function:test,
$T, $E)
```

et pas le code la déclarant :

```
$code Get-Content MonScript.ps1
[System.Management.Automation.Language.Parser]::ParseInput($code, $T, $E)
```

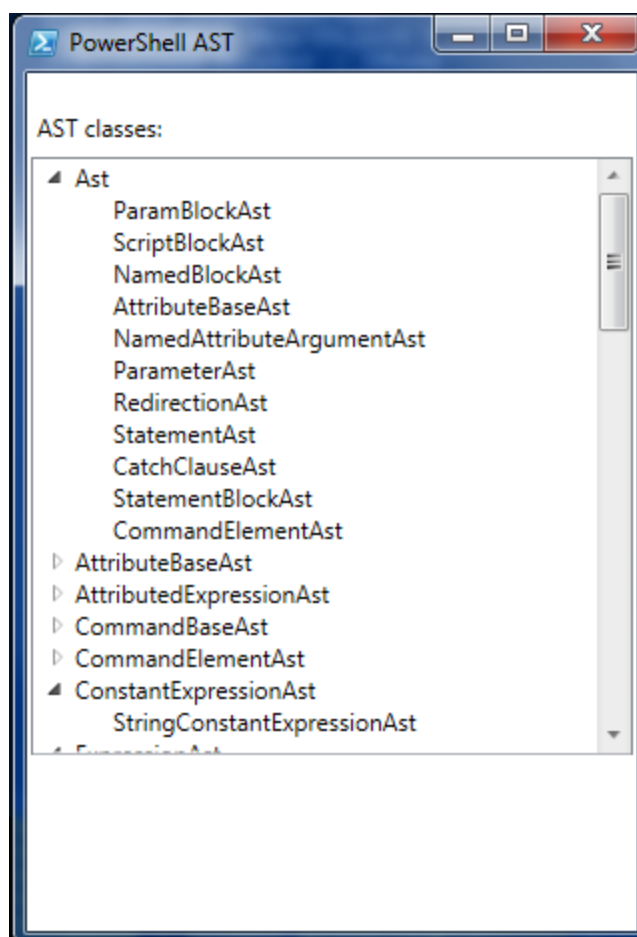
Pour lister les classes AST on doit d'abord retrouver l'assembly les hébergeant :

```
$Asm=[System.AppDomain]::CurrentDomain.GetAssemblies() |  
where {$_.ManifestModule.Name -eq 'System.Management.Automation.dll'}
```

La convention de nommage des classes permet de filtrer celles publiques :

```
$Asm.GetExportedTypes() |  
where {$_.Name -match 'Ast$'} |  
select name,basetype |  
sort basetype |  
fl name -groupby Basetype  
BaseType : System.Management.Automation.Language.CommandElementAst  
Name : CommandParameterAst  
Name : ExpressionAst  
...
```

Le script *Show-ClassName.ps1* propose un affichage dans un TreeView :





### 4.3 Au près de mon arbre

Pour une exploration rapide dans la console la navigation dans les nœuds peut se faire ainsi :

```
$Ast.EndBlock
```

```
Unnamed : True  
BlockKind : End  
Statements : {(3+5)*2+(6-3)}  
Traps :  
Extent : (3+5)*2+(6-3)  
Parent : (3+5)*2+(6-3)
```

```
$Ast.EndBlock.Statements|fl
```

```
PipelineElements : {(3+5)*2+(6-3)}  
Extent : (3+5)*2+(6-3)  
Parent : (3+5)*2+(6-3)
```

```
$Ast.EndBlock.Statements.PipelineElements|fl
```

```
Expression : (3+5)*2+(6-3)  
Redirections : {}  
Extent : (3+5)*2+(6-3)  
Parent : (3+5)*2+(6-3)
```

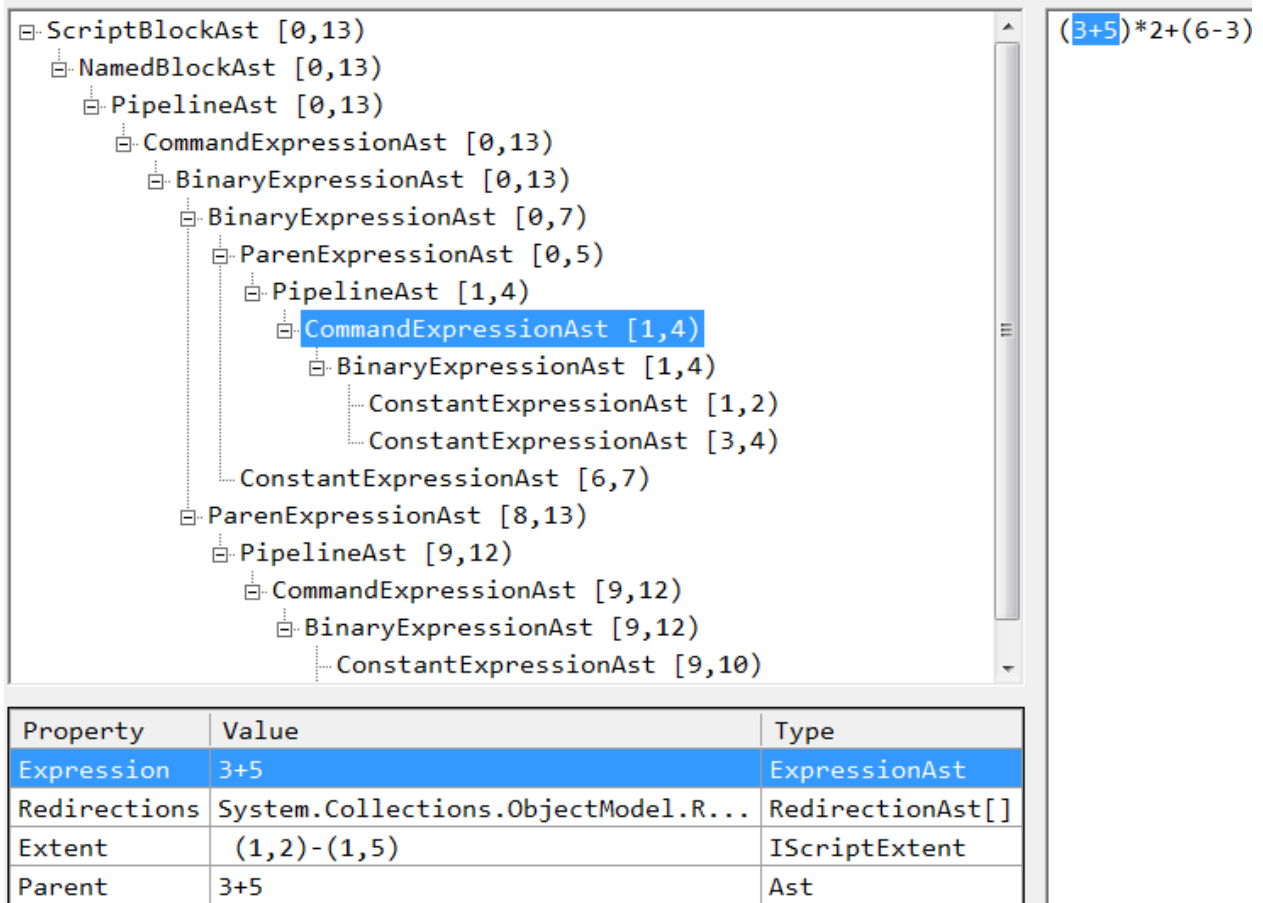
```
$Ast.EndBlock.Statements.PipelineElements.Expression|fl
```

```
Operator : Plus  
Left : (3+5)*2  
Right : (6-3)  
ErrorPosition : +  
StaticType : System.Object  
Extent : (3+5)*2+(6-3)  
Parent : (3+5)*2+(6-3)  
...
```

Cette approche est suffisante pour explorer UN nœud 'simple'.

On peut aussi utiliser le module nommé [Show-PSAST](#) proposant un GUI :

```
cd 'RépertoireDesSources\Modules'
Ipmo .\ShowAst\Show
Show-Ast $Ast
```



The screenshot shows a GUI window with a tree view on the left and a text area on the right. The tree view displays the AST structure for the expression  $(3+5)*2+(6-3)$ . The selected node is `CommandExpressionAst [1,4]`, which contains a `BinaryExpressionAst [1,4]` with two `ConstantExpressionAst` children: `ConstantExpressionAst [1,2]` and `ConstantExpressionAst [3,4]`. Below the tree is a table with the following properties:

Property	Value	Type
Expression	3+5	ExpressionAst
Redirections	System.Collections.ObjectModel.R...	RedirectionAst[]
Extent	(1,2)-(1,5)	IScriptExtent
Parent	3+5	Ast

L'usage du script *Show-Object.ps1* est une autre possibilité, celui-ci affiche le détail des membres d'un objet.

#### 4.4 Trouvez-les tous !

Pour retrouver tous les nœuds de l'arbre, la navigation de nœud en nœud dans la console est insuffisante.

Le parcourt des nœuds de l'arbre peut se faire à l'aide de la méthode *FindAll()* de la classe AST. Celle-ci, sans rentrer dans le détail de type, attend deux paramètres :

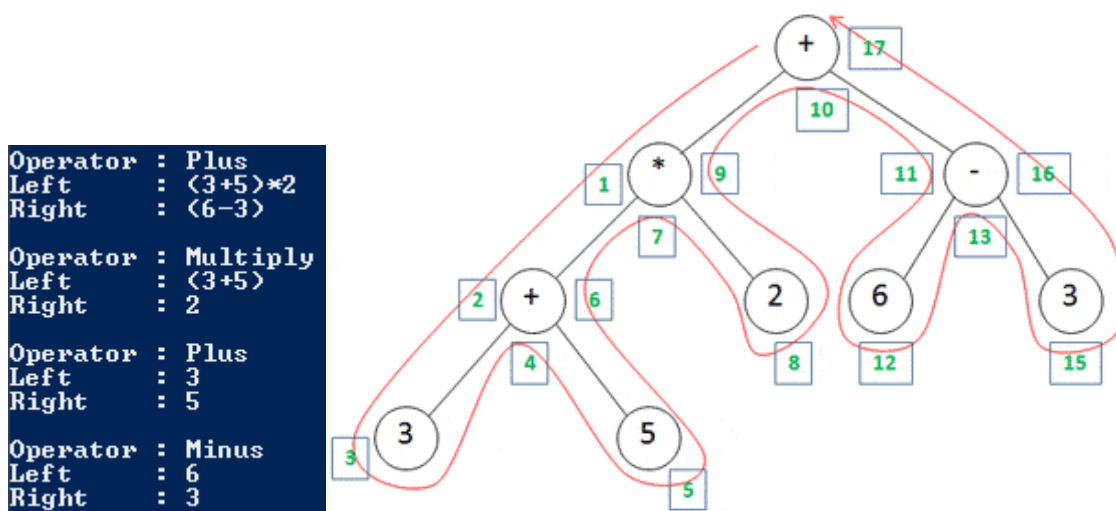
- un scriptblock qui doit renvoyer un booléen et
- un booléen indiquant si on parcourt les nœuds enfants. Tous les nœuds héritant de la classe AST, chaque nœud pourra être interrogé au cas par cas :

```
$Funcion= { $Args[0] -is  
[System.Management.Automation.Language.BinaryExpressionAst] }
```

*\$Args* contient le nœud courant reçu par la méthode *FindAll()*, le code du scriptblock retrouve, à partir de la racine, tous les nœuds de type *BinaryExpressionAst* :

```
$Ast.FindAll($Funcion, $true)|  
Select-Object Operator, Left, Right|  
Format-List
```

Le résultat nous affiche bien tous les éléments de notre expression :



<http://www.sunshine2k.de/coding/java/SimpleParser/SimpleParser.html>

Dans le schéma de droite, la 'ficelle' rouge indique l'ordre de lecture des nœuds de l'arbre, la première sous expression exécutée est (3+5).

Pour afficher tous les nœuds et faciliter l'exploration il suffit de ne pas filtrer sur le type des nœuds :

```
$Funcion= {Write-Warning "$($args[0].GetType().fullname) = $($Args[0])"}
$Ast.FindAll($Funcion, $true)
AVERTISSEMENT : System.Management.Automation.Language.ScriptBlockAst = (3+5)*2+(6-3)[0]
AVERTISSEMENT : System.Management.Automation.Language.NamedBlockAst = (3+5)*2+(6-3)[0]
AVERTISSEMENT : System.Management.Automation.Language.PipelineAst = (3+5)*2+(6-3)[0]
AVERTISSEMENT : System.Management.Automation.Language.CommandExpressionAst = (3+5)*2+(6-3)[0]
AVERTISSEMENT : System.Management.Automation.Language.BinaryExpressionAst = (3+5)*2+(6-3)[0]
...
```

La collection **\$TokensList** contient des objets *System.Management.Automation.Language.Token* et non plus *System.Management.Automation.PSToken* (Powershell v2) :

```
$TokensList.Count
0..13|
Foreach {
    $i=$_
    $TokensList[$i]; $TokensList[$i].Extent
    Read-Host "'Entrée' = suivant ; 'Control+c' = arrêter "
}
```

Cette fois la méthode *ParseInput* renvoie les résultats de l'analyse lexicale et syntaxique dans deux structures de données distinctes.

Ici aussi, pour l'expression "3+\*2", la collection **\$ErrorsAST** contiendrait les erreurs d'analyse :

```
$code='3+*2'
$ErrorsAst=$TokensList=$null
$Ast=[System.Management.Automation.Language.Parser]::ParseInput($Code, [ref]
)$TokensList, [ref]$ErrorsAst)
$ErrorsAst|fl
Extent      :
ErrorId     : ExpectedValueExpression
Message     : Vous devez indiquer une expression de valeur à droite de l'opérateur « + ».
IncompleteInput : False
```

#### 4.5 Un commentaire ?

L'AST ne contient pas les commentaires d'un code, les clauses telles que *#Requires* n'étant pas des mots clés du langage, mais des indications destinées au parseur, on y accèdera par la liste de tokens :

```
Text       : #Requires -Version 3.0
TokenFlags : ParseModeInvariant
Kind       : Comment
HasError   : False
Extent     : #Requires -Version 3.0
```

## 5 Une visite guidée.

Cette visite guidée explore un objet ou une structure de données. L'instance contenue dans la variable `$Ast` créée précédemment contient une méthode nommée `Visit()` :

```
$AST.Visit
OverloadDefinitions
-----
System.Object Visit(System.Management.Automation.Language.ICustomAstVisitor astVisitor)
void Visit(System.Management.Automation.Language.AstVisitor astVisitor)
```

Cette méthode est basée sur le design pattern *Visiteur* qui, pour faire simple, facilite le parcourt d'une structure de données dont les éléments sont de types différents (les classes xxxAST) et facilite également les modifications du code existant lors de l'ajout de nouvelles opérations (nouveaux mot clé du langage).

Nous utiliserons la seconde signature, celle ne renvoyant pas de résultat, la valeur du paramètre doit être une instance dérivée de la classe `System.Management.Automation.Language.AstVisitor`.

Ce qui nous contraint d'utiliser du code compilé, C# ou autre.

Note :

Le pattern utilisé par le runtime Powershell permet de parcourir tout ou partie de l'arbre, comme détaillé sur cette page : <http://c2.com/cgi/wiki?HierarchicalVisitorPattern>

### 5.1 Qu'est-ce que cette Visite ?

C'est une autre manière de parcourir les nœuds d'un arbre de syntaxe abstraite, elle nécessite de créer un visiteur, une classe implémentant tout ou partie des méthodes virtuelles de la classe [AstVisitor](#).

Précédemment avec la méthode `FindAll()`, c'est le scriptblock (**Powershell**) passé en paramètre qui déterminait le nœud à traiter. Avec la méthode `Visit()`, c'est le visiteur (**C#**) passé en paramètre qui sait comment traiter les nœuds de l'AST.

Chaque méthode de cette classe est dédiée à un type de nœud ce qui permet de spécialiser chaque traitement. L'AST propose un point d'entrée unique, mais on peut créer autant de classe visiteur que l'on veut.

Le visiteur sait comment manipuler les données reçues et seul le guide interne sait comment parcourir l'arbre. La visite peut se faire sur l'intégralité de l'arbre ou à partir d'un nœud.

## 5.2 Visiteur C#

L'exemple suivant construit une classe C#, son rôle est d'insérer dans une liste le nom de chaque paramètre de fonction rencontré dans l'AST :

```
$code=@'
using System.Collections.Generic;
using System.Management.Automation;
using System.Management.Automation.Language;

namespace MyVisitor
{
    public class ParameterVisitor : AstVisitor
    {
        // Liste des variables d'un bloc param
        public List<string> variables = new List<string>();

        public override AstVisitAction VisitParameter(ParameterAst
parameterAst)
        {
            //Variable param trouvée : ajout dans la liste
            variables.Add(parameterAst.Name.ToString());

            //On continue le parcourt de l'arbre
            return AstVisitAction.Continue;
        }
    }
}
'@

$asm=Add-Type -TypeDefinition $Code -Language CSharp -pass
#Code à analyser
$C=@'
function TokenTest{
    param ($A,
        [int] $B,
        [Parameter(Position=2, ParameterSetName="asObject")]
        [ValidateNotNull()]
        [AllowEmptyString()]
        [System.Management.Automation.PSObject] $InputObject )
}
'@
```

```

#contruction de l'AST du code précédent
[ref] $T=$null
[ref] $E=$null
$AST=[System.Management.Automation.Language.Parser]::ParseInput($C,$T,$E)

#Création du visiteur
$Visiteur=New-Object MyVisitor.ParameterVisitor
#Visite de l'arbre AST
$AST.Visit($Visiteur)
#Affiche le résultat
$Visiteur.variables
$A
$B
$InputObject

```

Pour tous les autres types de nœud leurs méthodes sont bien appelées, mais par défaut elles renvoient la valeur *AstVisitAction.Continue* sans effectuer de traitement.

### 5.3 *Visiteur basé module*

Pour un usage simple, le partage de données entre le code C# et la session Powershell peut s'avérer fastidieux, bien que le langage C# soit à privilégier pour ce type de manipulation.

Ceci dit, il existe une possibilité de coupler du code d'un module Powershell à une classe C# comme le fait le MVP Douglas Finke dans [un exemple](#) de son ouvrage (voir `chapter04\InvokeTemplate\Test-CodeAST.ps1`) :

*PowerShell for Developers*. Copyright 2012 O'Reilly Media, 978-1-4493-2270-0.

Il déclare une seule classe C# et autant de module que de visiteur. A l'usage on couple une instance de cette classe avec un module visiteur.

Son exemple étant basé sur du code Powershell dynamique, il vous faudra décomposer ses étapes pour retrouver le code de la solution.

Attention au contenu de la variable *\$OFS* qui peut provoquer des erreurs lors de la génération du code de classe C#, le code C# appelant la méthode *Invoke()* n'est pas protégé par un bloc `try/catch` et bien que la collection d'erreur *\$Error* soit renseignée les appels à **Write-Error** ne sont pas affichés sur la console.

## 5.4 Visiteur basé Scriptblock

De mon côté pour faciliter l'exploration rapide en ligne de commande, j'utilise également une classe C#, mais celle-ci appelle du code déclarée dans une hashtable, dont chaque clé est un nom de méthode de la classe *AstVisitor* et sa valeur un objet contenant le scriptblock à exécuter ainsi qu'une collection recevant le résultat de l'exécution du scriptblock.

Sa conception est similaire à l'approche basée module.

Le module **ExploreAST** n'est donc pas pour le moment un code de production. En revanche il est suffisant pour faciliter l'étude des classes AST ou effectuer des contrôles sur votre code à partir d'un poste de développement.

L'exemple précédent devenant :

```
Cd "Votre répertoireCodeSource"
Import-Module .\ExploreAST.psm1

$Code=@'
function TokenTest{
    param ($A,
        [int] $B,
        [Parameter(Position=2, ParameterSetName="asObject")]
        [ValidateNotNull()]
        [AllowEmptyString()]
        [System.Management.Automation.PSObject] $InputObject )
    }
'@
#Récupère l'AST
$AST= Get-AST -Input $Code

#Hashtable spécialisée portant les méthodes à exécuter
#lors du parcourt des nœuds de l'AST
$H=New-VisitorMembersDictionary

#Définition d'une méthode à l'aide d'un scriptblock
#Chaque nom de clé est un des noms de méthode de la classe AST
$H.VisitParameter={Param ($Ast) $Ast.Name.ToString()}
```

En interne Powershell convertit le scriptblock en une instance de la classe VisitorMembers :

```
$H.VisitParameter.GetType().FullName
Powershell.Visitor.VisitorMembers
```



Notre visiteur 'implémentera' une seule méthode :

```
$H.VisitParameter
Code          Result
----          -
Param ($Ast) $Ast.Name.ToString() {}

#Création du visiteur à partir de la hashtable
$Visiteur=New-AstVisitor $H
#Visite de l'arbre AST
#Renseigne la propriété 'Result' de la variable $Visiteur
Invoke-ScriptVisitor $AST $Visiteur

#Affiche pour chaque clé, la propriété 'Result' de
#la classe visitorMembers
Format-VisitorMembersDictionary $H

Key : VisitParameter
Result : {$A, $B, $InputObject}
Code : Param ($Ast) $Ast.Name.ToString()
```

Si l'instance du visiteur n'est pas nécessaire, on peut également utiliser cette fonction :

```
$H.VisitParameter.Result.Clear()
$Ast=Invoke-Visitors -InputScript $Code -visitorScript $H
$H.VisitParameter.Result
$A
$B
$InputObject
```

## 5.5 Limites

La construction de ces visiteurs se base sur l'appel à la méthode *Invoke()* d'un scriptblock ou d'un objet méthode de module. J'ai constaté les limites suivantes :

- Les appels à **Write-Error** renseigne la collection **\$Error**, mais leur affichage ne se fait pas sur la console, à la différence des exceptions et des autres flux (Debug, ...),
- La variable automatique *\$PSCmdlet* des fonctions avancées n'est pas renseignée (elle est à *\$null*).

Seul le dernier point concerne le code Powershell exécuté via un appel à *FindAll()*.

## 6 Script Analyzer

Cet [addon](#) pour ISE propose de contrôler votre code à l'aide de certaines règles (best practices).

Pour le moment (version 1.2.1) le nombre de règles est limité et leur exécution nécessite le chargement d'ISE, à moins d'utiliser ses classes C# comme dans le script *Test-ScriptRules.ps1* présent dans le répertoire des sources.

Vous constaterez dans le code du script que chaque classe implémentant une règle est en fait un visiteur héritant de la classe **AstVisitor**.

C'est un outil à suivre, car il se peut à l'avenir qu'il propose l'intégration de règles codées par nos soins et ainsi compléter les vérifications.

A noter que la fonction *Test-RuleEmptyCatchBlock* du module ExploreAST propose une implémentation en Powershell de la règle *CheckForEmptyCatchBlock* :

```
$Code=@'
try { dir }
catch {}
'@

$Ast= Get-AST -Input $Code
$Ast.FindAll({function:Test-RuleEmptyCatchBlock}, $true)
```

Il existe également le projet [PSReadLine](#) qui est un très bon exemple de ce que l'on peut faire autour de l'AST.

L'archive '*Script Line Profiler Sample.zip*', issue du SDK Powershell, contient un projet de cmdlet analysant le temps d'exécution des instructions d'un script.

## 7 Exemple : analyse d'une ligne de commande

Maintenant que nous avons la mécanique d'exploration on peut se pencher sur les traitements.

J'ai pris comme exemple la transformation d'un script de contrôle de code basé expressions régulières afin de les remplacer par l'AST.

La règle vérifie les clés de la hashtable utilisées par le cmdlet **Import-LocalizedData**, elle recherche les clés déclarées, mais inutilisées dans le code source, tout comme celles qui sont utilisées, mais pas déclarées dans la hashtable.

Le parsing de script ne pouvant considérer ces cas comme erronés, l'objectif est d'éviter des erreurs lors de l'exécution.

Le script d'origine se nomme '*Test-LocalizedData.ps1*' et la nouvelle version '*Test-LocalizedDataPSV3.ps1*'.

## 7.1 Principe

On doit dans un premier temps retrouver dans le code source toutes les lignes d'appel au cmdlet **Import-LocalizedData**, comme celle-ci par exemple :

```
Import-LocalizedData -BindingVariable ExploreAstMsgs `
                    -FileName ExploreASTLocalizedData.psd1
```

Puis, s'ils existent, extraire l'argument des paramètres '*BindingVariable*' et '*FileName*'.

Dans un second temps on recherche toutes les occurrences de la variable indiquée par le paramètre '*BindingVariable*', ici **ExploreAstMsgs**, afin d'extraire le nom de clé. Dans le code suivant il s'agit de **LocalizedDataKey** :

```
Write-Error $ExploreAstMsgs.LocalizedDataKey
```

Et enfin, comparer les noms de clé de la hashtable, contenus dans le fichier indiqué par le paramètre '*FileName*', ici le fichier **ExploreASTLocalizedData.psd1**, avec les noms de clé trouvés dans le code source.

Par défaut **Import-LocalizedData** recherche, à partir du répertoire du script courant, le fichier de localisation dans un sous-répertoire portant le nom de la culture courante.

Le résultat du traitement de recherche doit donc renvoyer suffisamment d'informations pour le traitement de comparaison :

```
$Module=import-module ExploreAST -Passthru
Test-LocalizedData -Path $Module.Path
FileName      : ExploreASTLocalizedData.psd1
BindingVariable : ExploreASTMsgs
ScriptName    : ExploreAST.psm1
BaseDirectory : C:\Temp\Module\ExploreAST
KeysFound     : {UnknownLocalizedDataKey, AliasParameterNotSupported, ... }
```

Note : bien que la convention d'usage veuille qu'un script ne contienne qu'un seul appel au cmdlet **Import-LocalizedData**, la fonction les prend tous en charge. Elle peut donc renvoyer une collection d'objets personnalisés.

Une fois ceci fait on peut intégrer ce script dans un mécanisme basé tâche tel que [Psake](#).

## 7.2 Contraintes

A partir de la version 3, le cmdlet autorise la construction suivante :

```
$ExploreAstMsgs = Import-LocalizedData `
                  -Filename ExploreASTLocalizedData.psd1
```

La ligne d'appel peut également utiliser certaines facilités du binding de paramètre, par exemple un nom de paramètre incomplet et des noms d'alias :

```
Import-LocalizedData -EA Stop -Binding ExploreAstMsgs `
                    -File ExploreASTLocalizedData.psd1
```

Les paramètres de type *Switch*, le plus souvent, n'ont pas de valeur et tous les paramètres acceptent la construction suivante :

```
Verb-Noun -ParameterName:Value
```

Voir :

<http://msdn.microsoft.com/en-us/library/system.management.automation.language.commandparameterast%28v=vs.85%29.aspx>

Le code doit également considérer ce cas :

```
-NomParamTypeSwitch ArgumentPositionel_1
```

Les arguments des paramètres doivent être des constantes de type *String*, les noms de variable sont bien récupérés, mais inutilisable dans un 'contexte d'analyse statique'.

Les types de construction étant nombreux, la recherche renvoie des nœuds AST, la récupération de l'information étant à la charge de l'appelant. Ceci évite une spécialisation de la fonction nommée *Get-CommandParameter*.

**Note** : On suppose que le code d'appel de ce cmdlet respecte les conventions d'usage. Par exemple on place le fichier de localisation dans un sous répertoire du module ou du script.

### 7.3 Limites

Dans cette implémentation ne sont pas pris en compte :

- Les paramètres positionnels,
- Les valeurs par défaut des paramètres,
- La liaison de paramètre via le pipeline,
- L'appel de la méthode de recherche de nom de paramètre attend un nom complet univoque (pas d'alias) : `Get-CommandParameter $Ast 'Import-LocalizedData' 'BindingVariable','FileName'`
- Les paramètres dynamiques ( <https://connect.microsoft.com/PowerShell/feedback/details/397832/> ),
- L'affectation multiple (`$a,$b=1,2` ou `$a=$b=3`),
- Les scriptblocks (*delayed script block binding*).

Certaines de ces limites s'expliquent par le contexte d'analyse qui est ici statique, par exemple la gestion du pipeline ou du contenu d'un nom de variable n'est possible que lors de l'exécution du script, tout comme l'usage de *\$PSDefaultParameterValues*.

D'autres sont des choix personnels, une règle de nommage/d'écriture d'appel du cmdlet

*Import-LocalizedData*, pouvant en annuler quelques-unes.

## 8 Exemple : Réécriture d'une instruction

Pour cet exemple j'ai choisi cette [optimisation de boucle](#). On recherche une instruction particulière, puis on la modifie si elle répond aux critères.

### 8.1 Principe

L'objectif est de transformer le code suivant :

```
$Range=1..10
For($i=0; $i -lt $Range.Count; $i++) { $i }
```

Par ce code :

```
$Range=1..10
$RangeCount = $Range.Count
For($i=0; $i -lt $RangeCount; $i++) { $i }
```

C'est une tâche de refactoring à implémenter sous ISE, mais pour ce tutoriel j'ai choisi de l'implémenter via une fonction. On pourrait ne pas implémenter la réécriture du code et s'arrêter au contrôle de ce type d'écriture (règle similaire à celles de l'addon *Script Analyzer*).

Les spécifications de la boucle for (*The for statement*) nous indiquent que l'expression qui nous intéresse ici se nomme *for-condition*, qu'elle supporte le pipeline et qu'elle est optionnelle.

Le traitement du visiteur consiste à tester les différents type de nœud que l'on retrouve dans l'AST d'une boucle *For*, puis à construire des objets contenant les informations de reconstruction de la boucle :

```
ID          : 4
Variable    : @ {Text=$RangeCount=$Range.Count; StartOffset=4; Length=-1}
Expression : @ {Text=$I -le $RangeCount; StartOffset=14; Length=25}
```

Ces informations sont utilisées par la fonction suivante :

```
function Set-AstForStatement {
    param(
        [Parameter(Mandatory=$true,ValueFromPipeline = $true)]
        [PSTypeName('ForStatementRewriter')]
        $InputObject,
        [Parameter(position=0,Mandatory=$true)]
        [System.Text.StringBuilder] $Code)
    process {
        $Pos=$InputObject.StartOffset
        if ( ($InputObject.Length -ne $null) -and ($InputObject.Length -ge 0) )
        { $Code.Remove($Pos,$InputObject.Length)> $null }
        $Code.Insert($Pos,$InputObject.Text) > $null
    }#process
}#Set-AstForStatement
```

## 8.2 Limites

Dans cette implémentation les trois écritures suivantes sont prises en compte :

```
$i -lt $Range.Count  
$i -lt $Range.Count-1  
$i -lt ($Range.Count-1)
```

mais pas celles-ci :

```
$i -lt (-1+$Range.Count)  
$Range.Count -gt $i
```

Ni toutes les autres, car si vous recherchez cette expression dans vos scripts vous constaterez que les trois premières formes sont les plus courantes.

## 9 Conclusion

L'usage de visiteur basé module est préférable ainsi que le refactoring *via* ISE qui permet de contrôler au fur et à mesure les modifications du code. Le mieux étant d'utiliser le C#.

La difficulté dans les premiers temps est de bien connaître les règles de syntaxe et de savoir les associer aux classes AST. L'implémentation du contrôle de règles simples est relativement aisée, par exemple celle d'un bloc try/catch vide, d'autres nécessiteront de nombreux efforts et certaines seront impossibles à coder, car l'AST est limité pour un langage dynamique.

La prise en compte des différentes syntaxes pour une même opération se posera souvent et parfois l'interrogation du ou des nœuds parents s'avérera nécessaire.

Des règles d'écriture pourront faciliter l'analyse de certaines lignes de code au détriment de leur rédaction. Par exemple le respect de la règle de l'addon ISE d'éviter l'usage de paramètre positionnel prend son sens ici.