Appendix C

WINDOWS PowerShell IN ACTION

SAMPLE CHAPTER

Bruce Payette

Foreword by Jeffrey Snover

MANNING

Windows PowerShell in Action

by Bruce Payette

Appenidx C

brief contents

APPENDIX C

The PowerShell grammar

One way to learn a new language is to look at its grammar. This appendix presents the PowerShell grammar annotated with notes and examples to help explain what's happening.

PowerShell is parsed using an augmented recursive descent parser. The augmentations are needed to deal with some of the complexities in tokenizing the PowerShell language. This topic is discussed in more detail in chapter 2.

The complete parser is composed of a set of parsing rules and tokenizing rules. These parsing and tokenization rules are what we're covering in this appendix. These rules can be separated into five layers:

1. Statement list: the rules for parsing a basic statement list and a statement block.
2. Statements: various kinds of statements in the language.
3. Expressions
4. Values
5. Tokenizer rules

In the following sections, we'll expand on each of these topics.

C.1 STATEMENT LIST

```
<statementBlockRule> =
  '{' <statementListRule> '}'

<statementListRule> =
  <statementRule> [ <statementSeparatorToken> <statementRule> ]*
```

C.2 STATEMENT

```
<statementRule> =
  <ifStatementRule> |
  <switchStatementRule> |
  <foreachStatementRule> |
  <forWhileStatementRule> |
  <doWhileStatementRule> |
  <functionDeclarationRule> |
  <parameterDeclarationRule> |
  <flowControlStatementRule> |
  <trapStatementRule> |
  <finallyStatementRule> |
  <pipelineRule>
```

C.2.1 Pipeline

```
<pipelineRule> =
  <assignmentStatement> | <firstPipelineElement> [ '|' <cmdletCall> ]*

<assignmentStatementRule> =
  <lvalueExpression> <AssignmentOperatorToken> <pipelineRule>

<lvalueExpression> =
  <lvalue> [? |? <lvalue>]*

<lvalue> =
  <simpleLvalue> <propertyOrArrayReferenceOperator>*

<simpleLvalue> =
  <AttributeSpecificationToken>* <variableToken>

<firstPipelineElement> =
  <expressionRule> | <cmdletCall>

<cmdletCall> =
  [ '&' | '.' | <empty> ] [ <name> | <expressionRule> ]
    [ <parameterToken> | <parameterArgumentToken> |
      <postfixOperatorRule> | <redirectionRule> ]*

<redirectionRule> =
  <redirectionOperatorToken> <propertyOrArrayReferenceRule>
```

Notes:

Here are examples showing what pipelines may look like:

```
  get-childitem -recurse -filter *.ps1 | sort name
  (2+3),3,4 | sort

  & "c:\a path\with spaces.ps1" | % { $_.length }
  get-childitem | sort-object > c:/tmp/junk.txt
```

This rule also handles parsing assignment expressions to allow things such as

```
  $a = dir | sort length
```

to parse properly.

C.2.2 The if statement

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

Notes:
The `if` statement is the basic conditional in Powershell. An example of an `if` statement is

```
if ($x -gt 100)
{
    "x is greater than 100"
}
elseif ($x -gt 50)
{
    "x is greater than 50"
}
else
{
    "x is less than 50"
}
```

In the PowerShell `if` statement, braces are required around the bodies of the statements even when the body is a single line. Also, the `elseif` token is a single word with no spaces. An `if` statement may return one or more values when used in a subexpression. For example:

```
  $a = $( if  ( $x –gt 100  ) { 100 } else { $x } )
```

will constrain the value of `$x` assigned to `$a` to be no larger than 100.

C.2.3 The switch statement

```
<switchStatementRule> =
  'switch' ['-regex' | '-wildcard' | '-exact' ]{0 |1}
    ['-casesensitive']{0|1}
    ['-file' <propertyOrArrayReferenceRule> |
        '(' <pipelineRule> ')' ]
       '{' [
        ['default' | <ParameterArgumentToken> |
          <propertyOrArrayReferenceRule> | <statementBlockRule> ]
        <statementBlockRule> ]+ '}'
```

Notes:
The `switch` statement allows you to select alternatives based on a set of clauses. It combines features of both the conditional and looping constructs. An example of a `switch` statement looks like

```
switch -regex -casesensitive (get-childitem | sort length)
{
    "^5" {"length for $_ started with 5" ; continue}
    { $_.length > 20000 } {"length of $_ is greater than 20000"}
    default {"Didn't match anything else..."}
}
```

There can only be one default clause in a `switch` statement

C.2.4 The foreach statement

```
<foreachStatementRule> =
  <LoopLabelToken>{0 |1} 'foreach' '(' <variableToken>
     'in' <pipelineRule> ')' <statementBlockRule>
```

Notes:
The `foreach` statement loops over an enumerable collection. An example of a `foreach` statement is:

```
foreach ($i in get-childitem | sort-object length)
{
    $i
    $sum += $i.length
}
```

Also note that there is a `Foreach-Object` cmdlet that can be used to process objects one element at a time. While this cmdlet is similar to the `foreach` statement, it is not part of the language.

C.2.5 The for and while statements

```
<forWhileStatementRule> =
  <LoopLabelToken>{0 |1} 'while' '(' <pipelineRule> ')'
    <statementBlockRule> |
 <LoopLabelToken>{0 |1} 'for' '(' <pipelineRule>{0 |1} ';'
    <pipelineRule>{0 |1} ';' <pipelineRule>{0 |1} ')'
       <statementBlockRule>
```

Notes:
A `while` statement looks like

```
while ($i -lt 100)
{
    echo i is $i
    $i += 1
}
```

A `for` statement looks like

```
for ($i=0; $i -lt 10; $i += 1)
{
     echo i is $i
}
```

C.2.6 The do/while and do/until statements

```
<doWhileStatementRule> =
  <LoopLabelToken>{0 |1} 'do' <statementBlockRule> ['while' | 'until']
    '('<pipelineRule> ')'
```

Notes:
Here is an example of a `do/while` statement:

```
do
{
    write-host $i
    $i += 1
} while ($i -lt 100)
```

And an example of a `do/until` statement:

```
do
{
    write-host $i
    $i += 1
} until ($i -ge 100)
```

C.2.7 The trap statement

```
<trapStatementRule> =
  'trap' <AttributeSpecificationToken>{0 |1} <statementBlockRule>
```

Notes:
A `trap` statement looks like

```
trap { ... }
```

or

```
trap [system.nullreferenceexception] { ... }
```

A `trap` statement is scoped to the statement list that contains it. See chapter 9.

C.2.8 The finally statement

```
<finallyStatementRule> =
  'finally' <statementBlockRule>
```

Notes:
A `finally` statement looks like

```
finally { ... }
```

This statement is not implemented in version 1 of PowerShell and will result in a not-implemented compile-time error.

C.2.9 Flow control statements

```
<flowControlStatementRule> =
  ['break' | 'continue']
    [<propertyNameToken> | <propertyOrArrayReferenceRule>]{0 |1} |
    'return' <pipelineRule>
```

Notes:
Flow control statements alter the normal flow of execution in PowerShell. Here are examples of what flow control statements look like:

```
   break
   break label
   break $labelArray[2].name
   return
   return 13
   return get-content | sort | pick-object -head 10
```

C.2.10 Function declarations

```
<functionDeclarationRule> =
  <FunctionDeclarationToken> <ParameterArgumentToken>
    [ '(' <parameterDeclarationExpressionRule> ')' ]
      <cmdletBodyRule>

<cmdletBodyRule> =
  '{' [ '(' <parameterDeclarationExpressionRule> ')' ] (
      [ 'begin' <statementBlock> |
        'process' <statementBlock> |
        'end' <statementBlock> ]* |
        <statementList> '}'
```

Notes:
Function declarations in PowerShell take a variety of forms, from a simple function with an implicit argument collection to a full cmdlet specification. A function definition in its simplest form looks like

```
    function foo { ... }
```

or

```
    function foo ($a1, $a2) { ... }
```

A function that acts like a full cmdlet looks like

```
    function foo ($a1, $a2) { begin { … } process { … } end { … } }
```

Parameters can alternatively be specified using the `param` statement:

```
    function foo ($a1, $a2) { begin { … } process { … } end { … } }
```

Finally, a filter may be specified as

```
filter foo  ( $a1, $a2 ) { … }
```

which is equivalent to

```
function ( $a1, $a2) { process { … } }
```

In all cases, the parameter specification is optional.

C.2.11 Parameter declarations

```
<parameterDeclarationRule> =
  <ParameterDeclarationToken> '('
    <parameterDeclarationExpressionRule> ')'

<parameterDeclarationExpressionRule> =
  <parameterWithIntializer>
    [ <CommaToken> <parameterWithIntializer> ]*

<parameterWithIntializer> =
  <simpleLvalue> [ '='  <expressionRule> ]
```

Notes:
This rule captures the parameter declaration notation in PowerShell. Parameter declarations allow for option type qualifiers and initializers to be specified for each parameter. Multiple type qualifiers can be specified to allow for more complex argument transformations. Argument initializers can contain subexpressions, allowing for arbitrary initialization including pipelines. Parameter declarations look like:

```
param ($x, $y)
param ([int] $a, $b = 13)
param ([int][char] $a = "x",
    [System.IO.FileInfo[]] $files = $(dir *.ps1 | sort length))
```

C.3 EXPRESSION

```
<expressionRule> = <logicalExpressionRule>

<logicalExpressionRule> =
  <bitwiseExpressionRule>
    [<LogicalOperatorToken> <bitwiseExpressionRule>]*

<bitwiseExpressionRule> =
<comparisonExpressionRule> [<BitwiseOperatorToken>
  comparisonExpressionRule>]*

<comparisonExpressionRule> =
  <addExpressionRule>
    [ <ComparisonOperatorToken> <addExpressionRule> ]*

<addExpressionRule> =
 <multiplyExpressionRule>
```

```
    [ <AdditionOperatorToken> <multiplyExpressionRule> ]*

<multiplyExpressionRule> =
  <formatExpressionRule>
    [ <MultiplyOperatorToken> <formatExpressionRule> ]

<formatExpressionRule> =
  <rangeExpressionRule>
    [ <FormatOperatorToken> <rangeExpressionRule> ]*

<rangeExpressionRule> =
  <arrayLiteralRule> [ <RangeOperatorToken> <arrayLiteralRule> ]*

<arrayLiteralRule> =
  <postfixOperatorRule> [ <CommaToken> <postfixOperatorRule> ]*

<postfixOperatorRule> =
  <lvalueExpression> <PrePostfixOperatorToken> |
  <propertyOrArrayReferenceRule>

<propertyOrArrayReferenceRule> =
  <valueRule> <propertyOrArrayReferenceOperator>*

<propertyOrArrayReferenceOperator> =
  '[' <expressionRule> ']' ] |
  '.' [ <PropertyNameToken> <parseCallRule>{0|1} | valueRule> ]

<parseCallRule> = '(' <arrayLiteralRule> ')'
```

C.4 VALUE

```
<valueRule> =
  '(' <assignmentStatementRule> ')' |
  '$(' <statementListRule> ')' |
  '@(' <statementListRule> ')' |
  <cmdletBodyRule> |
  '@{' <hashLiteralRule> '}' |
  <unaryOperatorToken> <propertyOrArrayReferenceRule> |
  <AttributeSpecificationToken> <propertyOrArrayReferenceRule> |
  <AttributeSpecificationToken> |
  <PrePostfixOperatorToken> <lvalue> |
  <NumberToken> |
  <LiteralStringToken> |
  <ExpandableStringToken> |
  <variableToken>

<hashLiteralRule> =
  <keyExpression> '=' <pipelineRule> [ <statementSeparatorToken>
    <hashLiteralRule> ]*
```

Notes:
The `valueRule` is used to process simple values where a simple value may actually include things such as hash literals or scriptblocks. This rule also handles the processing of subexpressions.

C.5 TOKENIZER RULES

```
<ComparisonOperatorToken> =
    "-eq" |  "-ne" |  "-ge" |  "-gt" |  "-lt" |  "-le" |
    "-ieq" | "-ine" | "-ige" | "-igt" | "-ilt" | "-ile" |
    "-ceq" | "-cne" | "-cge" | "-cgt" | "-clt" | "-cle" |
    "-like" | "-notlike" |  "-match" |  "-notmatch" |
    "-ilike" | "-inotlike" | "-imatch" | "-inotmatch" |
    "-clike" | "-cnotlike" | "-cmatch" | "-cnotmatch" |
    "-contains" |  "-notcontains" |
    "-icontains" | "-inotcontains" |
    "-ccontains" | "-cnotcontains" |
    "-isnot" | "-is" | "-as" |
    "-replace" | "-ireplace" | "-creplace"

<AssignmentOperatorToken> = "=" | "+=" | "-=" | "*=" | "/=" | "%="

<LogicalOperatorToken> = "-and" | "-or"

<BitwiseOperatorToken> = "-band" | "-bor"

<RedirectionOperatorToken> =
  "2>&1" | ">>" | ">" | "<<" | "<" | ">|" | "2>" | "2>>" | "1>>"

<FunctionDeclarationToken> = "function" | "filter"
```

An expandable string does variable expansion inside them.

```
<ExpandableStringToken> =  ".*"
```

A constant string doesn't do expansions; also escape sequences are not processed.

```
<StringToken> = '.*'
```

Variables look like `$a123` or `${abcd}` - escaping is required to embed { or } in a variable name.

```
<VariableToken> = \$[:alnum:]+  | \${.+}
```

The `ParameterToken` rule is used to match cmdlet parameters such as `-foo` or `-boolProp: <value>`. Note that this rule will also match `--foobar`, so this rule has to be checked before the `--token` rule.

```
<ParameterToken> = -[:letter:]+[:]{0 |1}

<CallArguementSeparatorToken> = ' |'

<CommaToken> = ' |'
```

```
<MinusMinusToken> = '--'
```

```
<RangeOperatorToken> = '..'
```

Tokenizing numbers is affected by what character follows them, subject to the parsing mode.

```
<NumberToken> = C# number pattern...
```

```
<ReferenceOperatorToken> = "." | "::" | "["
```

The following token rule is used to parse command argument tokens. It is only active after reading the command name itself. The goal is to allow any character in a command argument other than statement delimiters (newline, semicolon, or close brace), expression delimiters (close parenthesis, the pipe sysmbol), or whitespace. It's a variation of a string token (escaping works), but the token is delimited by any of a set of characters. The regular expression shown does not accurately capture the full details of how this works.

```
<ParameterArgumentToken> = [^-($0-9].*[^ \t]
```

```
<UnaryOperatorToken> = "!" | "-not" | "+" | "-" | "-bnot" | <attributeSpeci-
ficationToken>
```

```
<FormatOperatorToken> = '-f'
```

```
<LoopLabelToken> = [:letter:][:alnum:]*:
```

```
<ParameterToken> = "param"
```

```
<PrePostfixOperatorToken> = '++' | <MinusMinusToken>
```

```
<MultiplyOperatorToken> = '*' | '/' | '%'
```

```
<AdditionOperatorToken> = '+' | '-' | emDash | enDash | horizontalBar
```

The attribute specification looks like `[int]` or `[system.int32]` and will also eventually allow the full range of PowerShell metadata.

```
<AttributeSpecificationToken> =  \[..*\]
```

The following tokens make up the end-of-line token class. The tokens `&&` and `||` are not parsed but will result in a not-implemented error in version 1 of PowerShell.

```
<StatementSeparatorToken> = ';' | '&&' | '||' | <end-of-line>
```

A cmdlet name can be any sequence of characters terminated by whitespace that doesn't start with the one of the following characters. This basically matches a cmdlet that could be a native mode cmdlet or an executable name: foo/bar, foo/bar.exe, foo\bar.exe, foo.bar.exe, c:\foo\bar, and so on are all valid.

```
<CmdletNameToken> = [^$0-9(@"'][^ \t]*
```