RuleMethod

```
syntax RuleMethod         = RuleMethodAttributes "private"? Name "(" (RuleMethodsParam; ",")* ")" ":" 
Type RuleMethodBody;
syntax RuleMethodOverride = RuleMethodAttributes "override" Name RuleMethodBody;
syntax RuleMethodMissing  = RuleMethodAttributes "missing"  Name RuleMethodBody;
```

`RuleMethod` describes the methods defined directly in the body of the rule. Such methods can obtain information about the rule they defined in and also about its subrules. The methods can call for the other methods defined in the same rule as well as in other rules. In addition, the computation result of such methods can be cached directly in AST (created for the rules).

`RuleMethod` can manipulate subrule names to receive information about them and to call for their methods.

If `RuleMethod` is declared in `ExtensionRule`, a virtual method alike is created (in OOP terms). If this method doesn't have a body it will be automatically considered abstract, if it does then it will be considered virtual.

Virtual methods may (and abstract methods must) be redefined in the extensible rule extensions. This allows the author of the extensible rule to abstract the method from specified extensions, and lets the authors of the extensions create a method implementation specifically for their extensions.

When declaring an extension, the keyword override should be added. However, you should not specify the redefined method signature, as method overload is not supported.

`RuleMethod` allows you to perform any kind of calculations on the parsing result (AST). Using these calculations, you can carry out a multipass translation. Caching method results (marked with the `Cached` attribute) lets you split the processing into stages, since the values are saved directly in AST.

For example:

```
token Identifier = !(Keyword !IdentifierPartCharacters) IdentifierBody
{
  Value() : string = GetText(IdentifierBody);
}

syntax QualifiedIdentifier = Names=(Identifier; ".")+
{
  Parts() : list[string] = Names[0].Map(n => n.Value());
}
```

Here, `Value` returns the text, parsed by the `IdentifierBody` subrule of the `Identifier` rule, and the Parts method returns the list of the `QualifiedIdentifier` parts. `QualifiedIdentifier` consists of one or more identifiers divided by a period. The loop parsing result can be accessed through the Names field. Since this loop has a delimiter, its result is a two-element tuple.

The first tuple element contains an element list (their AST), the second – a list of delimiters (also their AST). The `Names0` construct provides an access to the first tuple element. Next, the standard Map function (similar to `Select` from Linq) transforms the AST elements list into a string list. The value of each element is calculated by calling the `Value` method, described in the `Identifier` rule.

Sometimes the parsed code is incorrect. In this case, during the error recovery, the parser may create surrogate AST branches for missing values. Upon a call to the surrogate AST method, an exception is issued. To prevent that, you can declare a special method form – `RuleMethodMissing`. This form cannot have any parameters – all it can do is return some value by default. This value will be used in case of a missing value.

Virtual method example:

```
token CharPart
{
  Value() : char;

  | Simple = !ReservedCharChar Char
    {
      override Value = FirstChar(this.Char);
    }
  | UnicodeEscapeSequence = "\\u" HexDigit HexDigit HexDigit HexDigit
    {
      override Value =
        HexToChar(this, HexDigit1.StartPos, HexDigit4.EndPos);
    }
  | EscapeSequence = "\\" Char
    {
      override Value = EscapeSequence(FirstChar(this.Char));
    }
}
```

And its usage:

```
token CharLiteral = "\'" CharPart "\'" { Value() : char = CharPart.Value(); }
```

`FirstChar` is an embedded method. It returns the first character parsed by the rule passed as a parameter.

`HexToChar` and `EscapeSequence` are regular methods, described in the same project (in a separate Nemerle-module):

```
public HexToChar(ast : Nitra.Ast, startPos : int, endPos : int) : char
{
  unchecked HexToInt(ast, startPos, endPos) :> char
}

public HexToInt(ast : Nitra.Ast, startPos : int, endPos : int) : int
{
  assert2(startPos < endPos);

  def text = ast.Location.Source.OriginalText;
  mutable result = HexDigit(text[startPos]);

  for (mutable i = startPos + 1; i < endPos; i++)
    unchecked result = (result << 4) + HexDigit(text[i]);

  result
}

public static EscapeSequence(c : char) : char
{
  | '\'' => '\'' | '\"' => '\"' | '\\' => '\\' | '0'  => '\0'
  | 'a'  => '\a' | 'b'  => '\b' | 'f'  => '\f' | 'n'  => '\n'
  | 'r'  => '\r' | 't'  => '\t' | 'v'  => '\v' | c    => c
}
```

See also

Name
RuleMethodAttributes
RuleMethodsParam
Type
RuleMethodBody