Functional parsers for an integration requests language grammar

Anton Antonov
MathematicaForPrediction project at GitHub
MathematicaForPrediction blog at WordPress
February 2014

Version 1.0

Introduction

The purpose of this slide show is to the give an introduction to using the package FunctionalParsers.m provided by the project Mathematica-ForPrediction at GitHub:

https://github.com/antononcube/MathematicaForPrediction/blob/master/FunctionalParsers.m
https://github.com/antononcube/MathematicaForPrediction

The functional parsers implementations are based on the article “Functional Parsers” by Jeroen Fokker.

The functional parsers presented form a monadic system. More explanations and references can be found at http://en.wikipedia.org/wiki/Parser_-combinator .

It is assumed the audience of this slide show is somewhat familiar with the parsing problem, the Backus-Naur Form, the concept of definite integrals in calculus.

This slide show is an aid for a recorded talk about functional parsers. (A podcast.)

I have the intent to update this slide show with clarification and extensions.

I have used the presented system of functional parsers to design and implement Domain-Specific Languages (DSL) and conversational engines for search and recommendation systems.

Document history

Version 1.0, 2014-02-02 .

Sentences

Suppose we want to make an interpreter of commands for single variable integrals calculation.

We can start by listing prototype sentences and then generalizing their form.

Here are some prototypes:

Calculate the integral of sin x over 0 to 1.

What is the integral of x^2+Log[x] for x in the interval 0 and 10?

Integrate Sin[x^2+5]

Integrate numerically 1/x+Sqrt[x] in the interval 4 and 10.

General command structure

We can say the integration commands have four parts:

1. type of integration (numerical or symbolical),

2. function to be integration (integrand),

3. variable (optional can be implied),

4. range (optional).

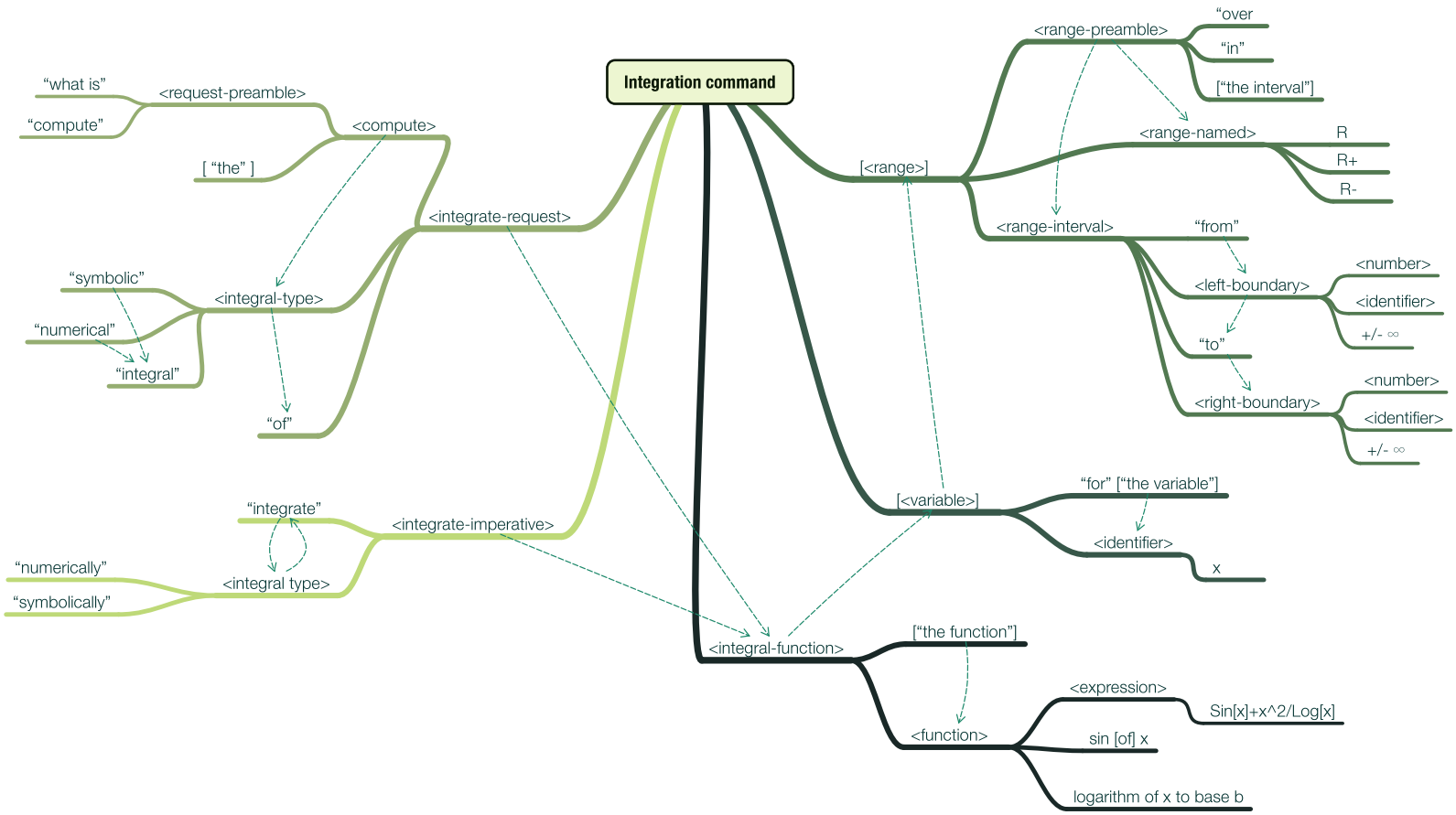
Roughly speaking we have the following structure:

<command preamble> <integral type> <integrand> <variable> <range>

Graph of the command structure

This graph represents the grammar of the integration commands we consider.

1. branching represents alternatives or composition,
2. the words and phrases and in quotes are tokens,
3. the abstract rules are in angular brackets (“<xxx>”),
4. the dashed arrows mean “followed by”,
5. square brackets (“[xxx]”) are for optional parts,
6. examples are given as “free” text.


Integration command
<request-preamble>
"what is"
"compute"
<compute>
["the"]
<integrate-request>
"symbolic"
"numerical"
<integral-type>
"integral"
"of"
"integrate"
<integrate-imperative>
"numerically"
"symbolically"
<integral type>
[<range>]
<range-preamble>
"over
"in"
["the interval"]
<range-named>
R
R+
R-
<range-interval>
"from"
<left-boundary>
<number>
<identifier>
+/- ∞
"to"
<right-boundary>
<number>
<identifier>
+/- ∞
[<variable>]
"for" ["the variable"]
<identifier>
x
<integral-function>
["the function"]
<function>
<expression>
Sin[x]+x^2/Log[x]
sin [of] x
logarithm of x to base b

How we do the parsing

We are going to program parsers for the grammar rules in the graph of the previous slide.

The graph follows closely the Extended Backus-Naur Form (EBNF) -- see http://en.wikipedia.org/wiki/Extended_Backus–Naur_Form .

Here is the EBNF for the integration command language we consider:

```
<request-preamble> = 'compute' | 'what' , 'is' ;
<compute> = <request-preamble> , [ 'the' ] ;
<integral-type> = [ 'numerical' | 'symbolic' ] , 'integral' ;
<integrate> = [ 'symbolically' | 'numerically' ] , 'integrate' | 'integrate' , ( 'numerically' |
'symbolically' ) ;
<function> = '_String' ;
<variable> = '_IdentifierString' ;
<range-named> =  'R' | 'R+' | 'R-' ;
<range-interval> = [ 'from' ] , '_WordString' , [ 'to' | 'and' ] , '_WordString';
<range> = ( [ 'in' | 'over' ] , [ 'the' ] , [ 'interval' ] ) , ( <range-interval> | <range-named> ) ;
<var-range-spec> = ( 'for' | 'of' ) , <variable> , <range> ;
<integral-request> = <compute> , <integral-type> , 'of' ;
<command1> = ( <integral-request> | <integrate> ) , <function> , ( <var-range-spec> | <range> ) ;
<command2> = ( <integral-request> | <integrate> ) , <function> ;
<command> = <command1> | <command2> ;
```

We are going to write a parser for each EBNF rule.

Later in the talk we are going to show that using the package FunctionalParsers.m we can generate the parsers from a string with the EBNF of the grammar.

Parsing <compute>

The first grammar rule we consider is <compute> that has the <request-preamble> as sub-part.

```
<request-preamble> = 'compute' | 'what' , 'is' ;
<compute> = <request-preamble> , [ 'the' ] ;
```

Taking LHS of rule we remove the dashes, "-", and put the prefix "p" in order to derive the symbol of parser corresponding to that rule.

Here is the parser for <request-preamble>:

```
pREQUESTPREAMBLE = ParseAlternativeComposition[ParseSymbol["compute"],
  ParseSequentialComposition[ParseSymbol["what"], ParseSymbol["is"]]]
```

```
ParseAlternativeComposition[If[Length[#1] > 0 && compute === First[#1], {{Rest[#1], compute}}, {}] &,
 ParseSequentialComposition[If[Length[#1] > 0 && what === First[#1], {{Rest[#1], what}}, {}] &,
  If[Length[#1] > 0 && is === First[#1], {{Rest[#1], is}}, {}] &]]
```

Which more compactly can be defines as:

```
pREQUESTPREAMBLE = ParseSymbol["compute"] ⊕ ParseSymbol["what"] ⊗ ParseSymbol["is"];
```

The symbol "⊕" is an infix notation shorthand of `ParseAlternativeComposition`.
The symbol "⊗" is an infix notation shorthand of `ParseSequentialComposition`.

Let us test the parser with the integration command string

```
icom = "what is the integral of Sin[x] over 0 1";
```

which we split into tokens first:

```
pREQUESTPREAMBLE[StringSplit[icom]]
```

```
{{{the, integral, of, Sin[x], over, 0, 1}, {what, is}}}
```

The parsing of <request-preamble> is successful, but in order to see than we need to explain the function parser definition.

Parser definition

First definition: We define a parser type to be a function that transforms a list of characters or tokens into (1) an unparsed list of tokens and (2) a parse tree:

```
P: _String → {_String, _ParseTree }

or

P: {_String...} → { {_String...}, _ParseTree }
```

It is better though to generalize this definition to allow a list of results:

```
P: {_String...} → { { {_String...}, _ParseTree }...}
```

(This is related to the so called “method of lists of success.”)

The functional parsers are categorized in the groups: basic, combinators, and transformers.
1. The basic parsers parse specified strings and strings adhering to predicates.
2. The combinator parsers allow sequential and alternative combinations of parsers.
3. The transformer parsers change the input or the output of the parsers that are transformed.

The parser `ParseSymbol` is a basic parser; `ParseAlternativeComposition` and `ParseSequentialComposition` are combinators.

Parsing <compute>, continued

So far we have the parsing result:

```
pREQUESTPREAMBLE[StringSplit[icom]]
```

```
{{{the, integral, of, Sin[x], over, 0, 1}, {what, is}}}
```

The parsing of <request-preamble> is successful. Looking at the grammar rules

```
<request-preamble> = 'compute' | 'what' , 'is' ;
<compute> = <request-preamble> , [ 'the' ] ;
```

we are ready to define <compute> that combines <request-preamble> with an optional “the”:

```
pCOMPUTE = pREQUESTPREAMBLE ⊗ ParseOption[ParseSymbol["the"]];
```

The definition of <compute> uses another combinator parser `ParseOption` for optional tokens.

The parsed result contains two possible parsings, one with “the“ parsed, and one without:

```
pCOMPUTE[StringSplit[icom]]
```

```
{{{integral, of, Sin[x], over, 0, 1}, {{what, is}, {the}}},
 {{the, integral, of, Sin[x], over, 0, 1}, {{what, is}, {}}}}
```

We can use the parser combinator ParseShortest to select the one parsed the token “the”:

```
ParseShortest[pCOMPUTE][StringSplit[icom]]
```

```
{{{integral, of, Sin[x], over, 0, 1}, {{what, is}, {the}}}}
```

Parsing <integral-type> and <integral-request>

The definition of the parser <integral-type>

```
pINTEGRALTYPE = ParseOption[ParseSymbol["numerical"] ⊕ ParseSymbol["symbolic"]] ⊗ ParseSymbol["integral"];
```

follows directly the EBNF rule

```
<integral-type> = [ 'numerical' | 'symbolic' ] , 'integral' ;
```

Now we can define <integral-request>

```
pINTEGRALREQUEST = pCOMPUTE ⊗ pINTEGRALTYPE;
```

```
icom
```

```
what is the integral of Sin[x] over 0 1
```

```
pINTEGRALREQUEST[StringSplit[icom]]
```

```
{{{of, Sin[x], over, 0, 1}, {{{what, is}, {the}}, {{}, integral}}}}
```

Picking only the important part

In many cases the semantic meaning of a grammar rule can be identified only by some part of the parsed input.
For example, in the context of the integration language the phrases “compute” and “what is” are needed in order to extend the number of admissible sentences, but as soon as they are parsed they are no longer of interest. What is of interest is to know what type of integration is requested: numerical or symbolical. (If the type is not explicitly specified we are going to interpret as “symbolic”.)

Instead of using `ParseSequentialComposition` (⊗) in the parser definition for the rule <integration-request> we can use `ParseSequentialCompositionPickRight` (▷) :

```
pINTEGRALREQUEST = pCOMPUTE ▷ pINTEGRALTYPE;
```

```
pINTEGRALREQUEST[StringSplit[icom]]
```

```
{{{of, Sin[x], over, 0, 1}, {{}, integral}}}
```

Compare with the parsing with the previous definition using `ParseSequentialComposition` (⊗) :

```
(pCOMPUTE ⊗ pINTEGRALTYPE)[StringSplit[icom]]
```

```
{{{of, Sin[x], over, 0, 1}, {{{what, is}, {the}}, {{}, integral}}}}
```

If the type of integration is explicitly specified as in

```
nicom = "what is the numerical integral of Sin[x] over 0 1";
```

then the new definition of <integral-request> outputs:

```
pINTEGRALREQUEST[StringSplit[nicom]]
```

```
{{{of, Sin[x], over, 0, 1}, {{numerical}, integral}}}
```

Transforming the parser results

Obviously we can (and should) go further and throw away also “integral“ using `ParseSequentialCompositionPickLeft` (◃) as in

```
pINTEGRALTYPE = ParseOption[ParseSymbol["numerical"] ⊕ ParseSymbol["symbolic"]] ◃ ParseSymbol["integral"];
pINTEGRALREQUEST = pCOMPUTE ▹ pINTEGRALTYPE;
```

```
incom
pINTEGRALREQUEST[StringSplit[incom]]
```

```
what is the numerical integral of Sin[x] over 0 1
```

```
{{{of, Sin[x], over, 0, 1}, {numerical}}}
```

But now if do not specify explicitly the integration type we are going to have an empty successfully parsed part:

```
icom
pINTEGRALREQUEST[StringSplit[icom]]
```

```
what is the integral of Sin[x] over 0 1
```

```
{{{of, Sin[x], over, 0, 1}, {}}}
```

Of course when we interpret the parsed output we can imply from no integral type specification that the integration is symbolic, but it is better if we can modify the parsed output of the rule <integral-type> to indicate the integration type.

This can be done using the parser transformer `ParseApply` (infix shorthand ⊙) that applies a function to the successfully parsed part.

```
pINTEGRALTYPE = ParseApply[IType,
   ParseOption[ParseSymbol["numerical"] ⊕ ParseSymbol["symbolic"]] ◃ ParseSymbol["integral"]];
pINTEGRALREQUEST = pCOMPUTE ▹ pINTEGRALTYPE;
```

```
pINTEGRALREQUEST[StringSplit[icom]]
```

```
{{{of, Sin[x], over, 0, 1}, IType[{}]}}
```

```
pINTEGRALREQUEST[StringSplit[nicom]]
```

```
{{{of, Sin[x], over, 0, 1}, IType[{numerical}]}}
```

Transforming the parser results, continued

Let us apply a more complicated function to the rule <integral-type>. For example,

```
pINTEGRALTYPE =
  ParseApply[
   IType[# /. {{} → "Symbolic", _ → "Numeric"}] &,
   ParseOption[ParseSymbol["numerical"] ⊕ ParseSymbol["symbolic"]] ◃ ParseSymbol["integral"]];
pINTEGRALREQUEST = pCOMPUTE ▷ pINTEGRALTYPE;
```

With the new definition we get

```
icom
pINTEGRALREQUEST[StringSplit[icom]]
```

```
what is the integral of Sin[x] over 0 1
```

```
{{{of, Sin[x], over, 0, 1}, IType[Symbolic]}}
```

```
nicom
pINTEGRALREQUEST[StringSplit[nicom]]
```

```
what is the numerical integral of Sin[x] over 0 1
```

```
{{{of, Sin[x], over, 0, 1}, IType[Numeric]}}
```

Other basic parsers

The only basic parse we used so far is `ParseSymbol`. A very often used one is `ParsePredicate`.

Suppose we want to parse a number string. Using `ParsePredicate` we can define the number parser as

```
pNumber = ParsePredicate[StringMatchQ[#, NumberString] &]
```

```
ParsePredicate[StringMatchQ[#1, NumberString] &]
```

`ParserPredicate` takes a function as an argument and the parsing is successful it that functions returns `True`.

```
pNumber[{"3432"}]
```

```
{{{}, 3432}}
```

Note that the parsed result is a string

```
% // FullForm
```

```
List[List[List[], "3432"]]
```

We can use `ParseApply` to convert to a number in *Mathematica*:

```
pNumber = ToExpression ⊙ ParsePredicate[StringMatchQ[#, NumberString] &]
```

```
ParseApply[ToExpression, ParsePredicate[StringMatchQ[#1, NumberString] &]]
```

```
pNumber[{"3432"}]
% // FullForm
```

```
{{{}, 3432}}
```

```
List[List[List[], 3432]]
```

Other basic parsers, continued

Here is the full list of the basic parser in the package FunctionalParsers.m :

`ParseSymbol[s]`	parses a specified symbol s.
`ParseToken[t]`	parses the token t.
`ParsePredicate[p]`	parses strings that give True for the predicate p.
`ParseEpsilon`	parses and empty string.
`ParseSucceed[v]`	does not consume input and always returns v.
`ParseFail`	fails to recognize any input string.

Other parser combinators

Here is a couple of more useful parser combinators: `ParseMany` and `ParseListOf`.

Suppose we want to parse sequence of numbers. It is easy to define a parser doing that using `ParseMany` and `ParsePredicate`:

```
pNumber = ToExpression ⊙ ParsePredicate[StringMatchQ[#, NumberString] &];
pNumberSeq = ParseMany[pNumber];
```

```
ParseJust[pNumberSeq][StringSplit["234 232 -532"]]
```

```
{{{}, {234, 232, -532}}}
```

We parsed with the parser transformer `ParseJust` in order to get results that parsed out the sequence completely.

If we want to parse a sequence of numbers separated by, say, semicolon we can use `ParseListOf`:

```
pNumberList = ParseListOf[pNumber, ParseSymbol[";"]];
```

```
ParseJust[pNumberList][StringSplit["234 ; 232 ; -532"]]
```

```
{{{}, {234, 232, -532}}}
```

Two other related parsers are `ParseChainLeft` and `ParseChainRight` :

```
ParseJust[ParseChainLeft[pNumber, ParseSymbol["+"]]][StringSplit["234 + 232 + -532"]]
```

```
{{{}, +[+[234, 232], -532]}}
```

```
ParseJust[ParseChainRight[pNumber, ParseSymbol["+"]]][StringSplit["234 + 232 + -532"]]
```

```
{{{}, +[234, +[232, -532]]}}
```

The rest of the integration grammar rules

At this point it should be clear how we proceed with defining the rest of the grammar rules of the integration request language.

In order to keep things simple the integration function is assumed to be a *Mathematica* expression with no spaces within it.

For example

```
ParseJust[pCOMMAND][StringSplit["integrate Sin[x]Log[x]/3 over 0 1"]]
{{{}, ICommand[{IType[{}], {IFunc[Sin[x]Log[x]/3], IRange[{0, 1}]}}]}}
```

The type of the integration is going to be implied to be symbolic if `IType` has an empty list.

For the grammar rules <variable> and <range> we use `IVar` and `IRange` respectively:

```
ParseJust[pCOMMAND][StringSplit["compute the numerical integral of Log[x] of x from 0 to 10"]]
{{{}, ICommand[{IType[{}], {IFunc[Log[x]], {IVar[x], IRange[{0, 10}]}}}]}}
```

Next we are going to generate automatically the parsers using the EBNF grammar.

Parser generation

Using the EBNF of the grammar of the integration commands language we can automatically generate the parsers for its grammar rules using the function `GenerateParsersFromEBNF` provided by the package FunctionalParsers.m .

In the current version of the EBNF parser functions there is a requirement that all elements of the parsed EBNF forms have to be separated by space.

The EBNF grammar string can have the pick-left and pick-right combinators (◁ and ▷ respectively) and a `ParseApply` specification can be given within the form “<rhs> = parsers <@ transformer” . The application of functions can be done over the whole definition of an EBNF non-terminal symbol, not over the individual parts.

Here is the grammar of the integration language:

```
integrationGrammar = "
<request-preamble> = 'compute' | 'what' , 'is' ;
<compute> = <request-preamble> , [ 'the' ] ;
<integral-type> = [ 'numerical' | 'symbolic' ] ◁ 'integral' <@ IType ;
<integrate> = [ 'symbolically' | 'numerically' ]
    ◁ 'integrate' | 'integrate' ▷ ( 'numerically' | 'symbolically' ) <@ IType ;
<function> = '_String' <@ IFunc ;
<var> = '_IdentifierString' <@ IVar ;
<range-named> =  'R' | 'R+' | 'R-' <@ IRange ;
<range-interval> = [ 'from' ] ▷ '_WordString' , [ 'to' | 'and' ] ▷ '_WordString' <@ IRange ;
<range> = ( [ 'in' | 'over' ] , [ 'the' ] , [ 'interval' ] ) ▷ ( <range-interval> | <range-named> ) ;
<var-range-spec> = ( 'for' | 'of' ) ▷ <var> , <range> ;
<integral-request> = <compute> ▷ <integral-type> ◁ 'of' ;
<command1> = ( <integral-request> | <integrate> ) , <function> , ( <var-range-spec> | <range> ) ;
<command2> = ( <integral-request> | <integrate> ) , <function> ;
<command> = <command1> | <command2> <@ ICommand ;
";
```

Parser generation, continued

Let us see how the parsed tree of the grammar looks like:

```
code = ToTokens[integrationGrammar];
Magnify[ParseEBNF[code], 0.7]
```

```
{{{}, EBNF[{EBNFRule[
     {{<request-preamble>, EBNFAlternatives[{EBNFSequence[EBNFTerminal['compute']], EBNFSequence[,[EBNFTerminal['what'], EBNFTerminal['is']]]]}}, ;}],
    EBNFRule[{{<compute>, EBNFAlternatives[
        {EBNFSequence[,[EBNFNonTerminal[<request-preamble>], EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['the']]}]]]]}}, ;}],
    EBNFRule[{{<integral-type>, EBNFAlternatives[{EBNFSequence[◃[EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['numerical']],
              EBNFSequence[EBNFTerminal['symbolic']]}]]], EBNFTerminal['integral']]]}]}, {<@, IType}}],
    EBNFRule[{{<integrate>, EBNFAlternatives[{EBNFSequence[◃[EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['symbolically']],
              EBNFSequence[EBNFTerminal['numerically']]}]]], EBNFTerminal['integrate']]], EBNFSequence[▹[EBNFTerminal['integrate'],
           EBNFAlternatives[{EBNFSequence[EBNFTerminal['numerically']], EBNFSequence[EBNFTerminal['symbolically']]}]]]]}], {<@, IType}}],
    EBNFRule[{{<function>, EBNFAlternatives[{EBNFSequence[EBNFTerminal['_String']]]}], {<@, IFunc}}],
    EBNFRule[{{<var>, EBNFAlternatives[{EBNFSequence[EBNFTerminal['_IdentifierString']]]}], {<@, IVar}}],
    EBNFRule[{{<range-named>, EBNFAlternatives[{EBNFSequence[EBNFTerminal['R']], EBNFSequence[EBNFTerminal['R+']], EBNFSequence[EBNFTerminal['R-']]]}],
      {<@, IRange}}], EBNFRule[{{<range-interval>,
       EBNFAlternatives[{EBNFSequence[▹[EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['from']]}]], ,[EBNFTerminal['_WordString'],
            ▹[EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['to']], EBNFSequence[EBNFTerminal['and']]}]], EBNFTerminal['_WordString']]]]]]}],
      {<@, IRange}}], EBNFRule[{{<range>, EBNFAlternatives[{EBNFSequence[▹[EBNFAlternatives[
            {EBNFSequence[,[EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['in']], EBNFSequence[EBNFTerminal['over']]}]], ,[EBNFOption[
                 EBNFAlternatives[{EBNFSequence[EBNFTerminal['the']]}]], EBNFOption[EBNFAlternatives[{EBNFSequence[EBNFTerminal['interval']]}]]]]]]}],
           EBNFAlternatives[{EBNFSequence[EBNFNonTerminal[<range-interval>]], EBNFSequence[EBNFNonTerminal[<range-named>]]}]]]]}], ;}],
    EBNFRule[{{<var-range-spec>, EBNFAlternatives[{EBNFSequence[▹[EBNFAlternatives[{EBNFSequence[EBNFTerminal['for']],
              EBNFSequence[EBNFTerminal['of']]}]], ,[EBNFNonTerminal[<var>], EBNFNonTerminal[<range>]]]]}], ;}], EBNFRule[
     {{<integral-request>, EBNFAlternatives[{EBNFSequence[▹[EBNFNonTerminal[<compute>], ◃[EBNFNonTerminal[<integral-type>], EBNFTerminal['of']]]]}],
      ;}], EBNFRule[{{<command1>, EBNFAlternatives[{EBNFSequence[
           ,[EBNFAlternatives[{EBNFSequence[EBNFNonTerminal[<integral-request>]], EBNFSequence[EBNFNonTerminal[<integrate>]]}], ,[EBNFNonTerminal[
              <function>], EBNFAlternatives[{EBNFSequence[EBNFNonTerminal[<var-range-spec>]], EBNFSequence[EBNFNonTerminal[<range>]]}]]]]]}], ;}],
    EBNFRule[{{<command2>, EBNFAlternatives[{EBNFSequence[,[EBNFAlternatives[{EBNFSequence[EBNFNonTerminal[<integral-request>]],
              EBNFSequence[EBNFNonTerminal[<integrate>]]}], EBNFNonTerminal[<function>]]]]}], ;}],
    EBNFRule[{{<command>, EBNFAlternatives[{EBNFSequence[EBNFNonTerminal[<command1>]], EBNFSequence[EBNFNonTerminal[<command2>]]}]}],
      {<@, ICommand}}]]}]}}
```

Parser generation, continued

Let us generate the grammars with `GenerateParsersFromEBNF`:

```
code = ToTokens[integrationGrammar];
res = GenerateParsersFromEBNF[code];
LeafCount[res]
```

```
822
```

and tabulate the parsed results of a set of commands with `ParsingTestTable`:

```
commands = {
    "compute the symbolic integral of Sin[x]*x^2 over 0 to 10",
    "what is the numerical integral of t^4+Log[t] for t in the interval 0 to 10",
    "integrate Sin[x]*x^2 over a b",
    "integrate Exp[-x] over R+"
   };
ParsingTestTable[ParseShortest[pCOMMAND], commands, "Layout" → "Vertical"] // Magnify[#, 0.8] &
```

1	command: compute the symbolic integral of Sin[x]*x^2 over 0 to 10 parsed: ICommand[{IType[symbolic], {IFunc[Sin[x]*x^2], IRange[{0, 10}]}}] residual: {}
2	command: what is the numerical integral of t^4+Log[t] for t in the interval 0 to 10 parsed: ICommand[{IType[numerical], {IFunc[t^4+Log[t]], {IVar[t], IRange[{0, 10}]}}}] residual: {}
3	command: integrate Sin[x]*x^2 over a b parsed: ICommand[{IType[{}], {IFunc[Sin[x]*x^2], IRange[{a, b}]}}] residual: {}
4	command: integrate Exp[-x] over R+ parsed: ICommand[{IType[{}], {IFunc[Exp[-x]], IRange[R+]}}] residual: {}

Interpretation

After we have generated the parsers of the integration requests grammar we have to write an interpreter of the parsed results.

Because of the way the parsing results are structured we can just write a definition for the function `ICommand` and the integrals are going to be computed during the parsing with `pCOMMAND`.

Instead, we are going to write a function `ICommandInterpreter` and use another function, `InterpretWithContext` (also provided by the package FunctionalParsers.m) to trigger the computation of the integrals.

```
ICommandInterpreter[parsed_] :=
  Block[{op, func, var, range},
   op = Flatten[Cases[parsed, IType[s_] :> s, ∞]];
   op =
    Which[
     MemberQ[op, "numerical"] || MemberQ[op, "numerically"], NIntegrate,
     True, Integrate
    ];
   func = ToExpression@Flatten[Cases[parsed, IFunc[s_] :> s, ∞]][[1]];
   var = Flatten@Cases[parsed, IVar[s_] :> s, ∞];
   If[Length[var] > 0,
    var = ToExpression@var[[1]],
    var = ToExpression@Flatten[Cases[func, s_Symbol /; ! NumericQ[s], ∞]][[1]]
   ];
   range = Flatten[Cases[parsed, IRange[s_] :> s, ∞]];
   If[Length[range] == 0,
    Integrate[func, x],
    range = Replace[range, {{"R"} -> {"-Infinity", "Infinity"}, {"R+"} -> {"0", "Infinity"}, {"R-"} -> {"-Infinity", "0"}}];
    range = ToExpression@range;
    op[func, Evaluate[Prepend[range, var]]]
   ]
  ];
```

Interpretation, continued

The function `InterpretWithContext` returns the interpretation of the input together with a list of rules of the context data if such data is given.

```
InterpretWithContext[
ParsingTestTable[ParseShortest[pCOMMAND], commands, "Layout" → "Vertical"],
  {ICommand -> ICommandInterpreter}] // Magnify[#, 0.8] &
```

1	command: compute the symbolic integral of Sin[x]*x^2 over 0 to 10 parsed: $-2 - 98\,\text{Cos}[10] + 20\,\text{Sin}[10]$ residual: {}
2	command: what is the numerical integral of t^4+Log[t] for t in the interval 0 to 10 parsed: 20 013. residual: {}
3	command: integrate Sin[x]*x^2 over a b parsed: $\left(-2 + a^2\right)\text{Cos}[a] - \left(-2 + b^2\right)\text{Cos}[b] - 2\,a\,\text{Sin}[a] + 2\,b\,\text{Sin}[b]$ residual: {}
4	command: integrate Exp[-x] over R+ parsed: 1 residual: {}

{ , {}}

Interpretation, continued

Here is a different set of integration commands:

```
commands = {
    "integrate symbolically Sin[x]*x^2 over a b",
    "compute the numerical integral of Sin[x] for x over 0 to 10",
    "integrate symbolically x+Sqrt[x] in the interval m0 and m1",
    "compute the integral of Log[x] of x from 0 to r"
  };
```

Now let us parse the integration commands using also data values for the context of interpretation.

```
InterpretWithContext[
ParsingTestTable[ParseShortest[pCOMMAND], commands, "Layout" → "Vertical"],
  {"data" → {a → A, b → B, m0 → 12, m1 → 100},
   "functions" → {ICommand -> ICommandInterpreter}}] // Magnify[#, 0.8] &
```

1	command: integrate symbolically Sin[x]*x^2 over a b parsed: $\left(-2+A^2\right)\text{Cos}[A]-\left(-2+B^2\right)\text{Cos}[B]-2\,A\,\text{Sin}[A]+2\,B\,\text{Sin}[B]$ residual: {}
2	command: compute the numerical integral of Sin[x] for x over 0 to 10 parsed: 1.83907 residual: {}
3	command: integrate symbolically x+Sqrt[x] in the interval m0 and m1 parsed: $\frac{16\,784}{3}-16\sqrt{3}$ residual: {}
4	command: compute the integral of Log[x] of x from 0 to r parsed: r (-1 + Log[r]) residual: {}

$\left\{\right.$, $\{a \to A, b \to B, m0 \to 12, m1 \to 100\}\left.\right\}$

Interpretation, continued

To summarize the interpretation:

1. the context is given as a list of rules;

2. there are two forms for the context rules: `{(_Symbol->_)..}` and `{"data"->{(_Symbol->_)...}, "functions"->{(_Symbol->_)...}}` ;

3. if during the interpretation the context functions change the context data the result of `InterpretWithContext` will return the changed data.

Discussion

The automatic generation of parsers from a grammar specification is very useful.

Large grammars would demand too much book keeping of the parsers for them. Automatic generation of parsers definitely helps.

As an example let us consider a change in the integration requests language to include the sub-command of plotting the integrand over the specified integration range. We want to be able to interpret sentences like these:

compute the integral of Log[x] of x from 0 to 10 and plot it

integrate and plot Sin[x]*x^2 over 0 1

We can just extend the grammar, re-generate the parsers, and provide interpretation for plotting clauses if they are present.

Here is the grammar with “plot it” added (the new symbol <plot> is used in <command1>):

```
integrationGrammar = "
<request-preamble> = 'compute' | 'what' , 'is' ;
<compute> = <request-preamble> , [ 'the' ] ;
<integral-type> = [ 'numerical' | 'symbolic' ] ◃ 'integral' <@ IType ;
<integrate> = [ 'symbolically' | 'numerically' ]
    ◃ 'integrate' | 'integrate' ▹ ( 'numerically' | 'symbolically' ) <@ IType ;
<function> = '_String' <@ IFunc ;
<var> = '_IdentifierString' <@ IVar ;
<range-named> =  'R' | 'R+' | 'R-' <@ IRange ;
<range-interval> = [ 'from' ] ▹ '_WordString' , [ 'to' | 'and' ] ▹ '_WordString' <@ IRange ;
<range> = ( [ 'in' | 'over' ] , [ 'the' ] , [ 'interval' ] ) ▹ ( <range-interval> | <range-named> ) ;
<var-range-spec> = ( 'for' | 'of' ) ▹ <var> , <range> ;
<integral-request> = <compute> ▹ <integral-type> ◃ 'of' ;
<plot> = 'plot' ◃ [ 'it' ] <@ IPlot ;
<command1> = ( <integral-request> | <integrate> , [ 'and' ▹
    <plot> ] ) , <function> , ( <var-range-spec> | <range> ) , [ 'and' ▹ <plot> ] ;
<command2> = ( <integral-request> | <integrate> ) , <function> ;
<command> = <command1> | <command2> <@ ICommand ;
";
```

Discussion, continued

And here is the interpretation function that both integrates and plots:

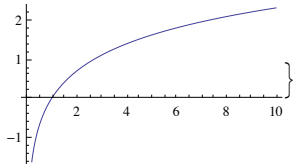
```
ICommandPlotInterpreter::noplot = "Cannot plot for a symbolic integral specification.";
ICommandPlotInterpreter[parsed_] :=
  Block[{op, func, var, range},
   op = Flatten[Cases[parsed, IType[s_] :→ s, ∞]];
   op =
    Which[
     MemberQ[op, "numerical"] || MemberQ[op, "numerically"], NIntegrate,
     True, Integrate
    ];
   func = ToExpression@Flatten[Cases[parsed, IFunc[s_] :→ s, ∞]][[1]];
   var = Flatten@Cases[parsed, IVar[s_] :→ s, ∞];
   If[Length[var] > 0,
    var = ToExpression@var[[1]],
    var = ToExpression@Flatten[Cases[func, s_Symbol /; ! NumericQ[s], ∞]][[1]]
   ];
   range = Flatten[Cases[parsed, IRange[s_] :→ s, ∞]];
   If[Length[range] == 0,
    If[Count[parsed, IPlot[_], ∞] > 0, Message[ICommandInterpreter::noplot]];
    {Integrate[func, x]},
    (*ELSE*)
    range = Replace[range, {{"R"} -> {"-Infinity", "Infinity"}, {"R+"} -> {"0", "Infinity"}, {"R-"} -> {"-Infinity", "0"}}];
    range = ToExpression@range;
    If[Count[parsed, IPlot[_], ∞] > 0,
     If[! VectorQ[range, NumberQ],
      Message[ICommandInterpreter::noplot];
      {op[func, Evaluate[Prepend[range, var]]]},
      {op[func, Evaluate[Prepend[range, var]]], Plot[func, Evaluate[Prepend[range, var]]]}
     ],
     (*ELSE*)
     {op[func, Evaluate[Prepend[range, var]]]}
    ]
   ]
  ];
```

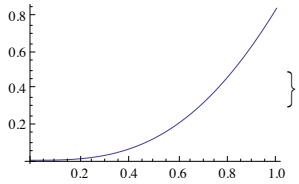
Discussion, continued

Let us try the new functionality out with several integration-and-plot commands:

```
commands = {
    "compute the integral of Log[x] of x from 0 to 10 and plot it",
    "integrate and plot Sin[x]*x^2 over 0 1",
    "integrate and plot Sin[x]*x^2 over a b"
  };
```

```
InterpretWithContext[
  ParsingTestTable[ParseShortest[pCOMMAND], commands, "Layout" → "Vertical"],
  {ICommand -> ICommandPlotInterpreter}] // Magnify[#, 0.8] &
```

1
command: compute the integral of Log[x] of x from 0 to 10 and plot it
parsed: $\{10\,(-1+\text{Log}[10]),$  $\}$
residual: {}

2
command: integrate and plot Sin[x]*x^2 over 0 1
parsed: $\{-2+\text{Cos}[1]+2\,\text{Sin}[1],$  $\}$
residual: {}

3
command: integrate and plot Sin[x]*x^2 over a b
parsed: $\{\left(-2+a^2\right)\text{Cos}[a]-\left(-2+b^2\right)\text{Cos}[b]-2\,a\,\text{Sin}[a]+2\,b\,\text{Sin}[b]\}$
residual: {}

$\{$ $, \{\}\}$

(We get the message “noplot” for the last integration request.)

Summary

We defined an EBNF grammar for an integration requests language and showed how parsers for it can be implemented using the package FunctionalParsers.m .

The system of parsers of the package has

1. Tokenizer
2. Elementary parsers
3. Parser combinators
4. Parser transformers
5. Binding for infix notation
6. Functions for automatic parser generation
7. A function for interpretation with context

We discussed the automatic parser generation and its usefulness for grammar extension and bookkeeping.