Parser combinator in R (*rmoparser*) : Vignette

February 5, 2026

Contents

1 Simple mathematical expressions

1.1 Prefix notation

Following Operator Precedence Parser entry in Wikipedia, we will construct a parser for simple mathematical expressions that turns infix notation into prefix notation. For example, given the expression

3 + 3

we will obtain

(+ (NUM 3) (NUM 3))

or given

x * sin(y)

we will obtain

(* (VAR x) (sin (VAR y)))

These prefix expressions can be easily evaluated to obtain their numerical value or to perform symbolic differentiation.

The grammar we will implement can be defined as follows:

```
list_expression = additive_expression, ws, ';', {
    additive_expression, ws, ';' } ;

additive_expression = ws, multiplicative_expression
    , { ws, ( '+' | '-' ), ws, multiplicative_
    expression } ;

multiplicative_expression = power_expression, { ws,
     ( '*' | '/' ), ws, power_expression } ;

power_expression = primary, [ ws, '**', ws, primary
     ] ;

primary = '(', ws, additive_expression, ws, ')' |
    '-', ws, primary | FUN, ws, '(', ws, additive_
    expression, ws, ')' | NUMBER | VARIABLE ;

NUMBER   = ? numberScientific ? ;

VARIABLE = ? symbolic ?         ;

FUN      = ? symbolic ?         ;

ws       = ? whitespace ?       ;
```

First rule, list_expression, implements recognition of a list of mathematical expressions delimited by semicolons ';'. The only action of this rule is printing the output.

```
> list_expression  <- function()
+ concatenation(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(s[[1]]))),
+ repetitionON(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(s[[1]])))),
+ action=function(s) NULL)
```

Second rule, additive_expression, implements sums and subtractions recognition, which are lower precedence operations. It generates a list following the pattern (+ operator1 operator2).

```
> library(qmrparser)
> additive_expression  <- function()  concatenation(ws(),multiplicative_expression(),
+ option( concatenation(ws(),
```

```
+ alternation(
+ keyword('+',action=function(s) s),
+ keyword('-',action=function(s) s),
+ action=function(s) s),
+ ws(),additive_expression(),
+ action=function(s) list(type='noempty',value=s[c(2,4)]))),
+ action=function(s) {if(s[[3]]$value$type=='empty') s[[2]] else
+ list(fun=s[[3]]$value$value[[1]],par1=s[[2]],par2=s[[3]]$value$value[[2]])
+ });
```

Next rule, multiplicative_expression, implements products and divisions recognition, which have higher precedence than sums and subtractions but lower than exponentiation or functions. It generates a list following the pattern (* operator1 operator2).

```
> multiplicative_expression  <- function()
+ concatenation(power_expression(),
+ option(
+ concatenation(ws(),
+ alternation(
+ keyword('*',action=function(s) s),
+ keyword('/',action=function(s) s),
+ action=function(s) s),
+ ws(),multiplicative_expression(),
+ action=function(s) {list(type='noempty',value=s[c(2,4)])})),
+ action=function(s) {
+ if(s[[2]]$value$type=='empty') s[[1]] else
+ list(fun=s[[2]]$value$value[[1]],par1=s[[1]],par2=s[[2]]$value$value[[2]])});
```

Rule power_expression recognises exponentiation. It generates a list following the pattern (ôperator1 operator2).

```
> power_expression  <- function()
+ concatenation(primary(),
+ option(
+ concatenation(ws(),keyword('**'),ws(),power_expression(),
+ action=function(s) list(type='noempty',value=s[[4]]))),
+ action=function(s){if(s[[2]]$value$type=='empty') s[[1]] else list(fun="^",par1=s[[
```

Lastly, basic elements are defined:

- Expression grouping within parenthesis
- Unary minus
- Functions

- Numbers
- Variables

```
> primary  <- function()  alternation(
+ concatenation(charParser('('),ws(),additive_expression(),ws(),charParser(')'),actio
+
+ concatenation(charParser('-'),ws(),primary(),
+ action=function(s) list(fun="U-",par1=s[[3]])),
+
+ concatenation(FUN(action=function(s) s), ws(), charParser('('), ws(), additive_expr
+ action=function(s) list(fun=s[[1]],par1=s[[5]])),
+
+ NUMBER  (action=function(s) list(fun="NUM", par1=s)),
+
+ VARIABLE(action=function(s) list(fun="VAR", par1=s)),
+ action=function(s) s);
```

Auxiliary functions which implement recognition for numbers, variables, function names or blanks.

```
> NUMBER    <- function(...)  numberScientific(...);
> VARIABLE  <- function(...)  symbolic(...);
> FUN       <- function(...)  symbolic(...);
> ws        <- function()     whitespace();
```

Moreover, in order to print in a clean way prefix notation:

```
> exprToString <- function(expr)
+ if ( !is.list(expr) ) as.character(expr) else
+ paste("(",paste(sapply(expr,exprToString,USE.NAMES=FALSE),collapse=" "),")")
```

And some examples:

```
> print("Infix to Prefix Examples")

[1] "Infix to Prefix Examples"

> invisible( list_expression()(streamParserFromString(" 8  ;")) )

[1] "( NUM 8 )"

> invisible( list_expression()(streamParserFromString("8 +4;")) )

[1] "( + ( NUM 8 ) ( NUM 4 ) )"

> invisible( list_expression()(streamParserFromString("8/2 ;")) )
```

```
[1] "( / ( NUM 8 ) ( NUM 2 ) )"

> invisible( list_expression()(streamParserFromString("8*2 ;")) )

[1] "( * ( NUM 8 ) ( NUM 2 ) )"

> invisible( list_expression()(streamParserFromString("2*3 + 4*5;")) )

[1] "( + ( * ( NUM 2 ) ( NUM 3 ) ) ( * ( NUM 4 ) ( NUM 5 ) ) )"

> invisible( list_expression()(streamParserFromString("sqrt( 16) ;")) )

[1] "( sqrt ( NUM 16 ) )"

> invisible( list_expression()(streamParserFromString("sin(3.1415) ;")) )

[1] "( sin ( NUM 3.1415 ) )"

> invisible( list_expression()(streamParserFromString("sin(3.14* (2*2+3+1)/2 ) ** 8;"

[1] "( ^ ( sin ( * ( NUM 3.14 ) ( / ( + ( * ( NUM 2 ) ( NUM 2 ) ) ( + ( NUM 3 ) ( NUM

> invisible( list_expression()(streamParserFromString("sqrt(16)**2+sin(3)-sin(3);"))

[1] "( + ( ^ ( sqrt ( NUM 16 ) ) ( NUM 2 ) ) ( - ( sin ( NUM 3 ) ) ( sin ( NUM 3 ) )

> invisible( list_expression()(streamParserFromString("sqrt(16)**2+sin(3)-sin(3)*2;")

[1] "( + ( ^ ( sqrt ( NUM 16 ) ) ( NUM 2 ) ) ( - ( sin ( NUM 3 ) ) ( * ( sin ( NUM 3

>
```

1.2 Calculator

In order to get a calculator, we only have to substitute the function which prints in a clean way by another which performs calculations:

```
> exprToNumber <- function(expr)
+ switch(expr[[1]],
+ 'NUM'= as.numeric(expr[[2]]),
+ 'VAR' =as.numeric(get(expr[[2]])),
+ 'U-'=-exprToNumber(expr[[2]]),
+ do.call(expr[[1]],unname(lapply(expr[-1],exprToNumber)))
+ )
```

and modify the list_expression function

```
> list_expression  <- function()
+ concatenation(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(exprToNumber(s[[1]])))),
+ repetitionON(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(exprToNumber(s[[1]]))))),
+ action=function(s) NULL);
```

Some examples:

```
> print("Calculator")
[1] "Calculator"
> invisible( list_expression()(streamParserFromString(" 8  ;")) )
[1] "8"
> invisible( list_expression()(streamParserFromString("8 +4;")) )
[1] "12"
> invisible( list_expression()(streamParserFromString("8/2 ;")) )
[1] "4"
> invisible( list_expression()(streamParserFromString("8*2 ;")) )
[1] "16"
> invisible( list_expression()(streamParserFromString("2*3 + 4*5;")) )
[1] "26"
> invisible( list_expression()(streamParserFromString("sqrt( 16) ;")) )
[1] "4"
> invisible( list_expression()(streamParserFromString("sin(3.1415) ;")) )
[1] "9.26535896604903e-05"
> invisible( list_expression()(streamParserFromString("sin(3.14* (2*2+3+1)/2 ) ** 8;"
[1] "2.71285778952491e-18"
> invisible( list_expression()(streamParserFromString("sqrt(16)**2+sin(3)-sin(3);"))
[1] "16"
> invisible( list_expression()(streamParserFromString("sqrt(16)**2+sin(3)-sin(3)*2;")
[1] "15.8588799919401"
>
```

1.3 Symbolic differentiation

In order to get a symbolic derivatives calculator, we only need to substitute the function which prints in a clean way by another function which performs differentiation.

Differentiation is performed in prefix notation and, as it is only an example, output is not printed in infix notation. Moreover, it is not implemented any set of functions neither simplification after differentiation.

```
> exprDeriv <- function(expr,var)
+ switch(expr[[1]],
+ 'NUM'= list("NUM", "0"),
+ 'VAR'  = if( expr[[2]] == var ) list("NUM" ,"1") else list("NUM", "0"),
+ "+"="-"= list(expr[[1]],exprDeriv(expr[[2]],var),exprDeriv(expr[[3]],var)),
+ "*"    =list("+",
+                list("*",expr[[2]],exprDeriv(expr[[3]],var)),
+                list("*",expr[[3]],exprDeriv(expr[[2]],var))
+           ),
+ "/"    =list("*",
+               list("-",
+                list("*",expr[[3]],exprDeriv(expr[[2]],var)),
+                list("*",expr[[2]],exprDeriv(expr[[3]],var))
+
+               ),
+               list("**",expr[[3]],"2")
+           ),
+ "sin"=list("*",exprDeriv(expr[[2]],var),list("cos", expr[[2]])),
+           list(paste("Diff",var,sep="_"),expr)
+ )
```

list_expression function must be modified:

```
> list_expression  <- function()
+ concatenation(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(exprDeriv(s[[1]],"x")))),
+ repetitionON(
+ concatenation(additive_expression(),ws(),keyword(';'),
+ action=function(s) print(exprToString(exprDeriv(s[[1]],"x"))))),
+ action=function(s) NULL);
```

Some examples:

```
> print("Differentiation")

[1] "Differentiation"
```

```
> invisible( list_expression()(streamParserFromString(" 8  ;")) )

[1] "( NUM 0 )"

> invisible( list_expression()(streamParserFromString(" x  ;")) )

[1] "( NUM 1 )"

> invisible( list_expression()(streamParserFromString("8 +x;")) )

[1] "( + ( NUM 0 ) ( NUM 1 ) )"

> invisible( list_expression()(streamParserFromString("x/2 ;")) )

[1] "( * ( - ( * ( NUM 2 ) ( NUM 1 ) ) ( * ( VAR x ) ( NUM 0 ) ) ) ( ** ( NUM 2 ) 2 )

> invisible( list_expression()(streamParserFromString("8*x ;")) )

[1] "( + ( * ( NUM 8 ) ( NUM 1 ) ) ( * ( VAR x ) ( NUM 0 ) ) )"

> invisible( list_expression()(streamParserFromString("2*x + 4*x;")) )

[1] "( + ( + ( * ( NUM 2 ) ( NUM 1 ) ) ( * ( VAR x ) ( NUM 0 ) ) ) ( + ( * ( NUM 4 )

> invisible( list_expression()(streamParserFromString("1+sqrt( x) ;")) )

[1] "( + ( NUM 0 ) ( Diff_x ( sqrt ( VAR x ) ) ) )"

> invisible( list_expression()(streamParserFromString("sin(x) ;")) )

[1] "( * ( NUM 1 ) ( cos ( VAR x ) ) )"

> invisible( list_expression()(streamParserFromString("sin(x* (2*2+x+1)/2 ) ** 8;"))

[1] "( Diff_x ( ^ ( sin ( * ( VAR x ) ( / ( + ( * ( NUM 2 ) ( NUM 2 ) ) ( + ( VAR x )

>
```

2 Wikipedia example

In Extended Backus–Naur Form can be found the following grammar definition example:

```
(* a simple program syntax in EBNF - Wikipedia *)
program    = 'PROGRAM' , white space , identifier ,
    white space ,
             'BEGIN' , white space ,
             { assignment , ";" , white space } ,
             'END.' ;

identifier = alphabetic character , { alphabetic
   character | digit } ;
number = [ "-" ] , digit , { digit } ;

string     = '"' , { all characters - '"' } , '"' ;

assignment = identifier , ":=" , ( number |
   identifier | string ) ;

alphabetic character = "A" | "B" | "C" | "D" | "E"
   | "F" | "G"
                      | "H" | "I" | "J" | "K" | "L"
                        | "M" | "N"
                      | "O" | "P" | "Q" | "R" | "S"
                        | "T" | "U"
                      | "V" | "W" | "X" | "Y" | "Z"
                        ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" |
   "7" | "8" | "9" ;

white space    = ? white space characters ? ;

all characters = ? all visible characters ? ;
```

Grammar definition is done in a language which has its own grammar. The wikipedia article does not make a formal definition of this grammar, but it does enummerate which symbols are used:

```
Table of symbols
The following represents a proposed standard.
Usage           Notation
definition      =
concatenation    ,
termination       ;
alternation     |
option          [ ... ]
repetition      { ... }
grouping        ( ... )
```

```
terminal string " ... "
terminal string ' ... '
comment          (* ... *)
special sequence ? ... ?
exception        -
```

The example developed next allows grammar recognition using these symbols and will recognise a simple program syntax in EBNF from Wikipedia example, while creating a parser for the defined grammar.

It is just a practical example to illustrate package function usage and it is not intended to recognise or parse arbitrary free context grammars. For this last purpose, the following references may be useful:

- Parsing
- Earley parser

A grammar is a collection/repetition of rules, ebnfRule. Some of these rules may be delimited by blanks and they end where the file containing grammar definition ends.

```
> gramatica <- function()
+ concatenation(
+ repetition1N(
+ concatenation(ebnfRule(),whitespace(),action=function(s) s[[1]]),
+ action=function(s) s),
+ eofMark(error=function(p) errorFun(p,h=NULL,type="eofMark")),
+ action=function(s) unlist(s[[1]]) )
```

A grammatical rule has a name, 'rule_name', a definition and ends with a semi-colon ';'

```
rule_name = definition ;
```

'rule_name' is a symbol, a sequences of letters, numbers and '_' beginning with a letter.

Therefore, rule definition is in parser notation:

```
symbolic(),charParser("="),ebnfDefinition(),
    charParser(';')
```

And taking into account blanks, concatenation and sequentially processing:

```
> ebnfRule <- function()
+ concatenation(
+
+ whitespace(),
```

```
+
+ symbolic(charFirst=isLetter,charRest=function(ch) isLetter(ch) || isDigit(ch) || ch
+
+ whitespace(),charParser("="),
+
+ whitespace(),ebnfDefinition(),whitespace(),charParser(';'),whitespace(),
+
+ action=function(s) paste(s[[2]]," <- function() ", s[[6]]))
```

A rule definition may be made up of just one or different possible alternatives.

```
> ebnfDefinition <- function() alternation(
+ # several alternatives
+ ebnfAlternation(),
+ # No alternatives
+ ebnfNonAlternation(),
+ action=function(s) s)
```

Alternatives making up a definition are delimited by "|"

```
> ebnfAlternation <- function()
+ concatenation(
+ ebnfNonAlternation(),
+
+ repetition1N(
+ concatenation(whitespace(),charParser("|"),whitespace(),ebnfNonAlternation(),action
+ action=function(s) paste("alternation(",paste(s[[1]],",",paste(unlist(s[[2]]), coll
```

The ways of rule definition which are not a list of alternatives is:

- A concatenation of definitions

  ```
  > ebnfConcatenation <- function()
  + option(
  + concatenation(
  + whitespace(),charParser(","),whitespace(),
  + ebnfNonAlternation(),
  + action=function(s) list(type="noempty",value=s[[4]])))
  ```

  this concatenation definition is added up to the other possibilities on the right side to avoid a problem with left recursive grammatical rules.

- String

```
> # string
> concatenation(
+ string(action=function(s) paste("keyword('",s,"')",sep="")),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
```

- A special sequence recognising non-string terminal symbols.

```
> # special sequence
> concatenation(
+ ebnfSpecialSequence(),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
```

- A symbol referencing a defined rule.

```
> # rule call
> concatenation(
+ symbolic(charFirst=isLetter,charRest=function(ch) isLetter(ch) || isDigit(ch)
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
```

- A parenthesised definition grouping other definitions

```
> # grouping
> concatenation(
+ whitespace(),charParser("("),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser(")"),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") s[[4]]
+ else paste("concatenation(",s[[4]],",",s[[7]]$value$value,")",sep=""))
```

- A definition repetition.

```
> # repetition
> concatenation(
+ whitespace(),charParser("{"),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser("}"),
```

```
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") paste("repetitionON(",s[[4]],")")
+ else paste("concatenation(",   paste("repetitionON(",s[[4]],")"),",", s[[7]]$v
```

- Optional application of a definition.

```
> # option
> concatenation(
+ whitespace(),charParser("["),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser("]"),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") paste("option(",s[[4]],")")
+ else paste("concatenation(",   paste("option(",s[[4]],")"),",", s[[7]]$value$v
```

Therefore, rule definition can be written as:

```
> ebnfAlternation <- function()
+ concatenation(
+ ebnfNonAlternation(),
+
+ repetition1N(
+ concatenation(whitespace(),charParser("|"),whitespace(),ebnfNonAlternation(),action
+ action=function(s) paste("alternation(",paste(s[[1]],",",paste(unlist(s[[2]]), coll
>  ebnfNonAlternation <- function() alternation(
+ # string
+ concatenation(
+ string(action=function(s) paste("keyword('",s,"')",sep="")),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
+                                                ,
+ # special sequence
+ concatenation(
+ ebnfSpecialSequence(),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
+                                                ,
+ # rule call
+ concatenation(
```

```
+ symbolic(charFirst=isLetter,charRest=function(ch) isLetter(ch) || isDigit(ch) || ch
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[2]]$value$type=="empty") s[[1]]
+ else paste("concatenation(",s[[1]],",",s[[2]]$value$value,")",sep=""))
+                                                ,
+ # grouping
+ concatenation(
+ whitespace(),charParser("("),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser(")"),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") s[[4]]
+ else paste("concatenation(",s[[4]],",",s[[7]]$value$value,")",sep=""))
+                                                ,
+ # repetition
+ concatenation(
+ whitespace(),charParser("{"),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser("}"),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") paste("repetition0N(",s[[4]],")")
+ else paste("concatenation(",   paste("repetition0N(",s[[4]],")"),",", s[[7]]$value$
+                                                ,
+ # option
+ concatenation(
+ whitespace(),charParser("["),whitespace(),
+ ebnfDefinition(),
+ whitespace(),charParser("]"),
+ ebnfConcatenation(),
+ action=function (s)
+ if(s[[7]]$value$type=="empty") paste("option(",s[[4]],")")
+ else paste("concatenation(",   paste("option(",s[[4]],")"),",", s[[7]]$value$value,
+                                                , action=function(s) s)
> ebnfDefinition <- function() alternation(
+ # several alternatives
+ ebnfAlternation(),
+ # No alternatives
+ ebnfNonAlternation(),
+ action=function(s) s)
```

'Special Sequence' definition will be associated with basic tokens.

```
> ebnfSpecialSequence <- function()
+ concatenation(whitespace(),charParser("?"),whitespace(),
+ alternation(
+ keyword("whitespace"      ,action=function(s) s),
+ keyword("symbolic"        ,action=function(s) s),
+ keyword("string"          ,action=function(s) s),
+ keyword("numberInteger"   ,action=function(s) s),
+ keyword("numberScientific",action=function(s) s),
+ action=function(s) paste(s,"()",sep="")),
+ whitespace(),charParser("?"),
+ action=function(s) s[[4]])
```

Finally, a function is needed to inform where the parser failed to recognise any text.

```
>   errorFun <- function(strmPosition,h=NULL,type="") {
+     if ( is.null(h) || type != "concatenation" )
+       print(paste("Error from line:",strmPosition$line,
+       " Caracter:",strmPosition$linePos," Stream Pos:", strmPosition$streamPos, "Ty
+     else errorFun(h$pos,h$h,h$type)
+
+     return(list(type=type,pos=strmPosition,h=h))
+   }
```

Some partial tests recognising simple rules:

1. ```
   > stream  <- streamParserFromString('program = \'PROGRAM\' ;')
   > cstream <- ebnfRule()(stream)
   > print(cstream[c("status","node")])

   $status
   [1] "ok"

   $node
   [1] "program  <- function()  keyword('PROGRAM')"
   ```

2. ```
   > stream <- streamParserFromString('program =  \'PROGRAM\' , white_space , ident
   > cstream <- ebnfRule()(stream)
   > print(cstream[c("status","node")])

   $status
   [1] "ok"

   $node
   [1] "program  <- function()  concatenation(keyword('PROGRAM'),concatenation(whit
   ```

3. ```
   > stream <- streamParserFromString(
   + '
   + program = \'PROGRAM\' , white_space , identifier , white_space ,
   +           \'BEGIN\'   , white_space ,
   +            { assignment , ";" , white_space } ,
   +            \'END.\' ;
   + ')
   > cstream <- ebnfRule()(stream)
   > print(cstream[c("status","node")])

   $status
   [1] "ok"

   $node
   [1] "program  <- function()  concatenation(keyword('PROGRAM'),concatenation(whit
   ```

4. ```
   > stream <- streamParserFromString(
   + 'identifier = alphabetic_character , { alphabetic_character | digit } ;')
   > cstream <- ebnfRule()(stream)
   > print(cstream[c("status","node")])

   $status
   [1] "ok"

   $node
   [1] "identifier  <- function()  concatenation(alphabetic_character(),repetition0
   ```

5. ```
   > stream <- streamParserFromString('white_space = ? whitespace ? ;')
   > cstream <- ebnfRule()(stream)
   > print(cstream[c("status","node")])

   $status
   [1] "ok"

   $node
   [1] "white_space  <- function()  whitespace()"
   ```

Now, the code will parse the whole set of rules from the Wikipedia example. They have been slightly modified in order to simplify programming:

```
> stream <- streamParserFromString(
+ '
+ program = \'PROGRAM\' , white_space , identifier , white_space ,
+           \'BEGIN\'   , white_space ,
+            { assignment , ";" , white_space } ,
```
```

```
+            \'END.\' ;
+ identifier = alphabetic_character , { alphabetic_character | digit } ;
+ number = [ "-" ] , digit , { digit } ;
+ assignment = identifier , ":=" , ( number | identifier | string_ ) ;
+ alphabetic_character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
+                      | "H" | "I" | "J" | "K" | "L" | "M" | "N"
+                      | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
+                      | "V" | "W" | "X" | "Y" | "Z" ;
+ digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
+ white_space = ? whitespace ? ;
+ string_ = ? string ?;
+ ')
> cstream <- gramatica()(stream)
> print(cstream[c("status")])

$status
[1] "ok"
```

As a result, we obtain functions in R capable of parsing texts in this grammar. R functions parse/eval are used to apply this parser.

```
> print(cstream[[c("node")]])

[1] "program  <- function()  concatenation(keyword('PROGRAM'),concatenation(white_spa
[2] "identifier  <- function()  concatenation(alphabetic_character(),repetitionON( al
[3] "number  <- function()  concatenation(option( keyword('-') ),concatenation(digit(
[4] "assignment  <- function()  concatenation(identifier(),concatenation(keyword(':='
[5] "alphabetic_character  <- function()  alternation(keyword('A'),keyword('B'),keywo
[6] "digit  <- function()  alternation(keyword('0'),keyword('1'),keyword('2'),keyword
[7] "white_space  <- function()  whitespace()"
[8] "string_  <- function()  string()"

> eval(parse(text=cstream[[c("node")]]))
```

We test some functions created in the previous step:

```
> identifier()(streamParserFromString("DEMO1"))$status

[1] "ok"

> identifier()(streamParserFromString("A0"))$status

[1] "ok"

> keyword(':=')(streamParserFromString(":="))$status
```

```
[1] "ok"

> number()(streamParserFromString("3"))$status

[1] "ok"
```

And finally, we will check that our parser can parse the program in the Wikipedia example.

```
> stream <- streamParserFromString(
+ 'PROGRAM DEMO1
+ BEGIN
+   A0:=3;
+   B:=45;
+   H:=-100023;
+   C:=A;
+   D123:=B34A;
+   BABOON:=GIRAFFE;
+   TEXT:="Hello world!";
+ END.')
> cstream <- program()(stream)
> if ( cstream$status=="fail" ) errorFun(cstream$node$pos,cstream$node$h,cstream$node

$status
[1] "ok"

>
```