“LaCoGen” – Lazarus Compiler Generator
User Documentation

Duncan Munro

31st January 2021 – v1.0

Synopsis: User instructions for operating the LaCoGen software.

Contents

1 Introduction

1.1 Outline

LaCoGen is a software tool to generate DFA (Deterministic Finite Automata) and LALR(1) Parser tables from a set of input rules, or grammar. Typical uses are language processors, scripting tools and compilers.

2 Quickstart

2.1 Generating the inputs

LaCoGen uses a plain text set of instructions, or grammar, stored in a .lac file. This file can be created with any suitable text editor.

2.2 Outputs from the tool

There are three possible outputs from LaCoGen:

- .lacobj file which is a compiled binary version of the DFA (Discrete Finite Automata) and LALR(1) (Look Ahead Left Right with 1 lookahead) outputs. The .lacobj file is used to drive a lexer and parser module later in the development cycle
- .txt file which is a textual representation of the grammar, its discrete elements and the outputs. This can run to many thousands of lines, however it does show many steps along the way such as the unoptimised non-deterministic finite automata (NFA)
- .xml file which is an experimental output in .xml. If nothing else, it's a human readable form of the DFA/LALR(1) output which can be imported into post-processing tools if you wish to do that

2.3 Running the Software

Simply type lacogen10 followed by the filename you wish to compile. Type the following to get more information on the command line parameters:

```
lacogen10 --help
```

This will provide a full list of all command line parameters available at this time.

3 Creating Your Application

Once you have created your grammar in a .lac file and compiled it into a .lacobj file, there are a number of steps to creating your application.

3.1 Required Files

Then construct an application containing the following two files from core/lac should be included in your Lazarus project:

- deployment_parser_module.pas
- deployment_parser_types.pas

The parser module constructs a parser object which can be used to load the .lacobj file either from a file or stream. It can also load from a resource which is the preferred method if your software is ready for production.

3.2 Other Steps

Your initialisation routine should do the following:

- Create an array of all the different rules
- Assign a function address to each one
- Create a dispatcher to take each reduction and call one of the functions from the array we just created

The example for the expression evaluator described in section 9 gives a more detailed view of how this takes place in practice.

4 Language definition

The tool is ultimately driven by grammar and is one component of a grammar processing system; the following high-level elements are used to make up a grammar:

- Parameters
- Character Sets
- Terminals
- Non-Terminals

Instruction lines are delimited with semicolon ; characters.

The following sections describe these five major building blocks in more detail.

5 Parameters

Parameters are used to direct the operation of the software. With the exception of the Start parameter, they are optional.

The general format is:

```
%Parameter-name = Parameter-value;
```

It is normal to place these are the start of the file for readability and values may be Boolean, Integer, String or Non-Terminal.

A complete list of parameters is given in Appendix A.

5.1 Boolean Parameters

Boolean parameters will have the value TRUE or FALSE (case is not important), for example:

```
%COMMENTNESTED = True;
```

5.2 Integer Parameters

Integer parameters will have a value typically in a 16 bit range. The appendix provides a list of acceptable limits for the values. Typical examples are:

```
%LEXERBUFFER = 4096;
%LEXERTAB = 4;
```

Out of range values will result in an error message.

5.3 String Parameters

String parameters are enclosed in quotes, typical examples are:

```
%AUTHOR = "John Doe";
%TITLE = "Expression Evaluator";
%VERSION = "2.0";
```

5.4 Non-Terminal Parameters

There is only one non-terminal parameter, the mandatory start parameter. This indicates the top of the grammar tree or the starting rule:

```
%START = <Content>;
```

Without this parameter, LaCoGen will raise an error and all processing will stop.

6 Character sets

Character sets are a set of zero or more characters in the following format:

```
{character_set_name} = [characters];
```

Note that the description is *zero or more* characters, an empty set is legal however its application is limited in this context. As can be seen, it's simply a name in curly braces followed by the equals sign and a set of characters in square brackets.

Valid definitions could be:

- [abcdefghijklmnopqrstuvwxyz] for lower case letters
- [0123456789ABCDEFabcdef] for hex digits
- [01] for binary digits

Ranges can be specified by using the hyphen:

- [a-zA-Z] for the alphabet
- [0-9] for decimal digits

These can save time and improve readability.

Multi-byte character sets are supported. Both these and other special characters can be specified by using a # or & symbol followed by the appropriate digits. # specifies a decimal number, whereas & specifies a hexadecimal number.

For example:

- [#9#10#13] contains Tab, Newline and Carriage return
- [&20-&7E] contains printable characters
- [&20AC] contains the Euro sign

Care needs to be exercised with these in respect of the surrounding context. While the start of the numeric specifier is well defined by the symbols & and #, the end is not.

For example [&20] is perfectly clear, however [&20ab] is ambiguous as it could be the character number &20 followed by ab or it could be the character number &20ab.

6.1 Character set escaping

It follows that some characters cannot be represented within the square brackets, those being [ ] # & -.

The backslash can be used to escape the next character from any rules, for example \[or \& are valid. It follows therefore that the backslash itself needs to be escaped which is done with \\.

For familiarity with the C language, the following escape sequences are also allowed:

- \t tab
- \n newline
- \r carriage return

6.2 Character set operators

Character sets permit some basic operations to add or remove items from a character set, these being + to add a set and – to remove one. For example:

{digits} = [0-9]
{letters} = [a-zA-Z}
{valid_sym_chars} = {digits} + {letters} + [_-$]

{printable} = [&20-&7E]
{escaped_chars} = [\[\&\#\-\]\\]
{allowed_chars} = {printable} – {escaped_chars}

7 Terminals

Terminals are the backbone of the lexical analyser. They are items which can be described as a sequence of characters and can be represented by a token.

7.1 Terminal Definition

A terminal is made up of a sequence of one or more terminal elements. A terminal definition will be:

```
Terminal-name = Terminal-content [optional keyword] ;
```

Where Terminal-content is one or more terminal elements. A terminal element can be:

- String
- Character set name
- Character set literal
- Another terminal already defined
- A bracketed terminal content

Examples are:

```
Keyword_FOR = "FOR";
Token_Digit = {Digit};
Terminator = ("END" | "EXIT") {NewLine};
```

7.2 Terminal Closures

Kleene Closures can be applied to the terminal content, the following are defined:

- Content ? – Zero or one instances of Content is allowed
- Content * - Zero or more instances of Content are allowed
- Content + - One or more instances of Content are allowed

Examples are:

```
StringLiteral = [“] {Valid-String-Character}* [“];
VariableName = {letter} ({alphanumeric} | “_”)*;
```

7.3 Terminal Concatenation

To concatenate terminal elements, simply list them one after the other as in the the example shown in sections 7.1 or 7.2.

7.4 Terminal Alternation

The alternation character is | and is used to denote a choice, for example:

Boolean = “TRUE” | “FALSE”;

7.5 Terminal Functions

Limited functions are available:

- ( ) parenthesis for grouping, e.g. “A” | “B” “C”. This gives A or B followed by C while “A” | (“B” | “C”) gives A followed by either B or C
- MIXED(string-value) generates a mixed case terminal. The terminal MIXED(“True”) will match True, TRUE or true. This is a shortcut and efficient way to implement mixed case keywords

7.6 Terminal Precendence

The terminal elements will be processed using the following four precedences from highest to lowest:

- Parenthesis, MIXED() function, existing terminal name, string literal, set name, set literal
- Closure, e.g. {digit}*
- Concatenation
- Alternation

If equal precendences are present, the evaluation will take place from left to right.

7.7 Terminal Keywords

An optional keyword can be placed at the end of each terminal definition to further instruct LaCoGen on the purpose of the terminal. The options are:

- Ignore – Do not use this terminal if present in the input. Typically this will be used for comments, whitespace and other items which are to be discarded rather than processed
- Keyword – Indicates that this is a keyword which will allow the DFA / Lexer to prioritise it as such. This allows the lexer to separate out FOR or NEXT as a keyword which avoids the ambiguity of deciding if the token is a keyword or a variable name
- Symbol – Indicates that the terminal is a symbol. It processes it also as a keyword, however it sorts it into a different area in the terminal table for better readbility

8 Non-Terminals

Non-terminals are items which need to be reduced, unlike terminals. The names are enclosed in angle brackets to define them as non-terminals, and they are typically used to construct rules. Each non-terminal has a definition with that definition consisting of terminals or non-terminals.

8.1 Rules

Typically the non-terminal definition is a rule or set of rules typically like the following:

```
<add-operation>     : <expression> "+" <expression>
                    | <expression> "-" <expression>
                    ;
```

As can be seen, the rule making up the definition for a non-terminal can be made up from non-terminals or terminals. Alternation can be used to indicate different rules which make up the non-terminal.

8.2 Epsilon Condition

A special case is called the Epsilon condition where an element making up the non-terminal could be completely empty. For example:

```
<opt-expr>      : <expression>
                |
                ;
```

Notice that nothing follows the final alternation character. This allows <opt-expr> to be completely empty.

8.3 Code Naming

Each rule will trigger a corresponding action which is called when the rule is reduced. LaCoGen will generate function names for each rule, however you can generate your own by putting # and a function name on the end of the rule. For example:

```
<add-op>  : <expr> "+" <expr>      # ActAdd
          | <expr> "-" <expr>      # ActSub
          | <expr>                 # ActCopy
          ;
```

As can be seen, there are three different functions for the three different rules that make up the non-terminal <add-op>. One useful instance for this naming is with non-terminals that don't actually do anything as they reduce. It's possible to create a single function called ActIgnore and place it against multiple rules. This will save a large number of empty functions being required.

9 Example Software

In the test folder is a floating point expression evaluator. Full source code along with the .lac files is provided. This is a good place to start if you want to understand how the software operates and the components which make it up.

9.1 Example External Files

The example uses the following items from LaCoGen:

- test_expr_fp.lacobj – compiled file which was created from the grammar presented in test_expr_fp.lac
- lac/core/deployment_parser_module.pas – Core file which defines the lacogen lexer and parser. This will load the .lacobj file to make a complete parser
- lac/core/deplopyment_parser_types.pas – File you can reference in your application with important type definitions that will be needed in functions your write

9.2 Example Outline Operation

The test example, through the Form Creation function in ftestexprfp.as creates an array called ReduceProcs which is a list of all the reduction actions for each reduction or "rule".

It then populates it one by one by registering each reduction procedure name and associated function.

Note that in the parser setup section it sets the following callbacks:

- OnMonitor – called by the parser when it wants to tell you something or has a problem
- OnReduce – called by the parser when it wants to reduce a rule. You have to provide the code to do this

9.3 Examining Rule Processing

Let's look at a rule from the .lac file and see how it's dealt with:

```
<Expression> : <Expression> "+" <AddOp> # procadd
```

The rule consists of the non-terminal <Expression> and three possible elements. We've told the software that we want it to call a function called procadd when we are reducing this rule.

The processing for this has already been registered as described in section 9.2:

```
MakeSetProc('procadd', @procadd);
```

Upon a reduction for this rule being required, the parser will look for the OnReduce callback. This goes to TForm1.Reduce() which then checks the array and passes over to the function procadd() we set earlier.

procadd() is in the file test_expr_fp_procs.pas and the code is:

```
function procadd(Parser: TLCGParser): TLCGParserStackEntry;
begin
  Result.Buf   := FloatToStr(StrToFloat(Parser.ParserStack[Parser.ParserSP-3].Buf) +
  StrToFloat(Parser.ParserStack[Parser.ParserSP-1].Buf));
end;
```

The following notes should be observed:

- All data is passed round as strings to avoid typing issues
- The function should return a TLCGParserStackEntry. If it has nothing to send back, it should set the .Buf element of the result to an empty string
- The incoming parser stack is referenced by ParserSP (Parser Stack Pointer)
- If there are 3 elements in the reduction, as with our example, they can be accessed in the order
    - ParserStack[ParserSP-3]
    - ParserStack[ParserSP-2]
    - ParserStack[ParserSP-1]
- Note that the addition rule has only two operands, however the ParserSP-2 element is the token for the "+" sign as there are three elements in total

Appendix A. Parameters

Parameter	Type	Description	Notes
%AUTHOR	String	Author of the grammar	Doesn't affect software operation, simply copied into .lacobj file
%CODEPREFIX	String	Prefix string to be used on any code block names. Allows multiple instances / lacogen objects in the same application	LaCoGen internally uses a parser for the grammar and a sub-parser for set literals. This allows the two to coexist and prevent them overwriting each other
%COMMENTNESTED	Boolean	Decides if block comments can be nested or not. For example the /* comments used in C	Can be TRUE or FALSE
%COPYRIGHT	String	Copyright message	Doesn't affect software operation, simply copied into .lacobj file
%LEXERBUFFER	Integer	Number of bytes used in the lexer buffer	Defaults to 4096 if not specified
%LEXERTAB	Integer	Number of characters represented by Horizontal Tab (CHR 9). Used for expansion of whitespace within the lexer	Defaults to 4 if not specified, typically 4 or 8
%LICENCE	String	Licence terms (abbreviated)	Doesn't affect software operation, simply copied into .lacobj file
%PARSERBUFFER	Integer	Number of bytes in the parser buffer for holding tokenized input	Defaults to 1024 if not specified
%START	Non-terminal	The non-terminal which is the start of the grammar definition	Mandatory. Enclosed in angle brackets as with

			other non-terminal names
%TITLE	String	Title of the LaCoGen grammar	
%UNITLEXER	String		Deprecated
%UNITPARSER	String		Deprecated
%VERSION	String	Version of the grammar file or application it's used in	Doesn't affect software operation, simply copied into .lacobj file

Appendix B. Grammar file for LaCoGen 1.0 (Main Grammar)

```
//------------------------------------------------------------------------------------
//
//  LaCoGen V1.0 - Grammar definition
//
//  04/04/2020
//
//  Items over V0.5 grammar include:
//
//    Parameter definition keywords removed
//    Block comments and line comments commands
//
//  Current list of parameters allowed with [default] if set:
//
//    Boolean       %COMMENTMATCH [True] +
//                  %COMMENTNESTED [True] +
//                  %COMMENTTOKENISE [True] +
//                  %GENERATEINDENTS [False] +
//
//    String        %AUTHOR
//                  %COMMENTBLOCK +#
//                  %COMMENTLINE +
//                  %COPYRIGHT
//                  %LICENSE
//                  %LICENSEURL +
//                  %TITLE
//                  %VERSION
//
//    Non-Terminal  %START
//
//  + Items new to version 1.0
//  # has two string arguments, e.g. %COMMENTBLOCK "/*" "*/"
//
// -----------------------------------------------------------------------------------

%title          = "LaCoGen Grammar";
%version        = "1.0";
%author         = "Duncan Munro";
%start          = <Content>;


// ======================================================================
// Predefined Character sets
// ======================================================================

{Alphanumeric}  = [0-9A-Za-z];
{Letter}        = [A-Za-z];
{Number}        = [0-9];
{Printable}     = [ -\xff];
{WSChar}        = [\t\r\n ];


// ======================================================================
// Character sets
// ======================================================================

{Escape}            = [\x5c];  // Backslash
{Quote}             = ["];
{EscapedChars}      = [tnrx\\\-\[\]];
{NonTerminal Ch}    = {Alphanumeric} + [_\-.] + [ ];
{Set Literal Ch}    = {Printable} - [\]] - {Escape};
{Set Name Ch}       = {Printable} - [{}];
{String Ch}         = {Printable} - {Quote};
{Terminal St}       = {Letter};
{Terminal Ch}       = {Alphanumeric} + [_\-.];
{Zero}              = [0];
{Nonzero Digit}     = {Number} - {Zero};

{Compilable}        = {Printable} + {WSChar};
```

```
{Non Asterisk}      = {Compilable} - [*];
{Non SlashAst}      = {Compilable} - [*/];
{Non Linebreak}     = {Compilable} - [\r\n];


// ======================================================================
// Terminals
// ======================================================================

TNonTerminal            = "<" {NonTerminal Ch}+ ">";
Terminal                = {Terminal St} {Terminal Ch}*;
SetLiteral              = "[" ({Set Literal Ch} | ({Escape} {EscapedChars}) )* "]";
SetName                 = "{" {Set Name Ch}+ "}";
StringLiteral           = ["] ({String Ch} | {Escape} {EscapedChars})* ["];
Integer                 = "-"? ({Zero} | {Nonzero Digit} {Number}*);
ParameterName           = "%" {Terminal St} {Terminal Ch}*;
KTrue                   = ("T"|"t") ("R"|"r") ("U"|"u") ("E"|"e") keyword;
KFalse                  = ("F"|"f") ("A"|"a") ("L"|"l") ("S"|"s") ("E"|"e") keyword;
F_MIXED                 = ("M"|"m") ("I"|"i") ("X"|"x") ("E"|"e") ("D"|"d") "(" keyword ;

BlockComment            = "/*" ( {Non Asterisk}* [*]+ {Non SlashAst} )* {Non Asterisk}* [*]+
"/" ignore;
LineComment             = "//" {Non Linebreak}* ([\r] [\n] | [\r] | [\n]) ignore;
WhiteSpace              = {WSChar}+ ignore;


// ======================================================================
// Rules
// ======================================================================

// High level rules

<Content>           : <Content> <BoundedDefinition>             # ActIgnore
                    | <BoundedDefinition>                       # ActIgnore
                    ;

<BoundedDefinition> : <Definition> ";"                          # ActIgnore
                    ;

<Definition>        : <ParameterDef>                            # ActIgnore
                    | <CharacterSetDef>                         # ActIgnore
                    | <TerminalDef>                             # ActIgnore
                    | <RuleDef>                                 # ActIgnore
                    ;


// Rules for Parameters

<ParameterDef>      : ParameterName "=" <BooleanTerminal>       # ActParameterDefBoolean
                    | ParameterName "=" Integer                 # ActParameterDefInteger
                    | ParameterName "=" StringLiteral StringLiteral # ActParameterDefString2
                    | ParameterName "=" StringLiteral           # ActParameterDefString
                    | ParameterName "=" TNonTerminal            # ActParameterDefNonTerminal
                    ;

<BooleanTerminal>   : KTrue                                     # ActCopy
                    | KFalse                                    # ActCopy
                    ;


// Rules for Character Sets

<CharacterSetDef>   : SetName "=" <CSContent>                   # ActSetDefine
                    ;

<CSContent>         : <CSContent> "+" <CSDef>                   # ActSetAdd
                    | <CSContent> "-" <CSDef>                   # ActSetSub
                    | <CSDef>                                   # ActSetUse
                    ;

<CSDef>             : SetName                                   # ActCopy
```

```
                    | SetLiteral                                  # ActSetLiteral
                    ;


// Rules for Terminals

<TerminalDef>       : Terminal "=" <TerminalContent> "ignore"   # ActTerminalDefIgnore
                    | Terminal "=" <TerminalContent> "keyword"  # ActTerminalDefKeyword
                    | Terminal "=" <TerminalContent> "symbol"   # ActTerminalDefSymbol
                    | Terminal "=" <TerminalContent>            # ActTerminalDef
                    | Terminal "virtual"                        # ActTerminalDefVirtual
                    ;

<TerminalContent>   : <TerminalContent> "|" <TerminalElement>   # ActTerminalOr
                    | <TerminalElement>                         # ActCopy
                    ;

<TerminalElement>   : <TerminalElement> <TerminalClosed>        # ActTerminalConcatenate
                    | <TerminalClosed>                          # ActCopy
                    ;

<TerminalClosed>    : <TerminalBracketed> "?"                   #
ActTerminalClosedBracketedOpt
                    | <TerminalBracketed> "+"                   #
ActTerminalClosedBracketedPlus
                    | <TerminalBracketed> "*"                   #
ActTerminalClosedBracketedStar
                    | <TerminalBracketed>                       # ActCopy
                    ;

<TerminalBracketed> : "(" <TerminalContent> ")"                 # ActTerminalBracketBracket
                    | F_MIXED StringLiteral ")"                 # ActTerminalBracketMixed
                    | Terminal                                  # ActTerminalBracketTerminal
                    | StringLiteral                             # ActTerminalStringLiteral
                    | SetName                                   # ActTerminalBracketSetName
                    | SetLiteral                                # ActTerminalBracketSetLiteral
                    ;


// Main rules

<RuleDef>           : <NonTerminal> ":" <RuleList>              # ActRuleDef
                    ;

<RuleList>          : <RuleList> "|" <Rule>                     # ActRuleListOrRule
                    | <RuleList> "|" <OptProc>                  # ActRuleListOrOptProc
                    | <Rule>                                    # ActCopy
                    ;

<Rule>              : <RuleBody> <OptProc>                      # ActRuleRuleBodyOptProc
                    ;

<RuleBody>          : <RuleBody> <RuleAtom>                     # ActRuleBodyRuleAtom
                    | <RuleAtom>                                # ActCopy
                    ;

<RuleAtom>          : <NonTerminal>                             # ActCopy
                    | Terminal                                  # ActRuleAtomTerminal
                    | StringLiteral                             # ActRuleAtomStringLiteral
                    ;

<NonTerminal>       : TNonTerminal                              # ActRuleAtomNonTerminal
                    ;

<OptProc>           : "#" Terminal                              # ActOptProcHashTerminal
                    |                                           # ActIgnore
                    ;
```

Appendix C. Grammar file for LaCoGen 1.0 (Set Literals)

```
//-------------------------------------------------------------------------------------
//
// LaCoGen - Grammar definition for set literals
//
//
// Examples are:
//
//   [0123456789]  // The digits
//   [0-9]         // Digits done a different way
//   [^0-9]        // Anything but digits
//   [A-Za-z]      // Letters
//   [\t]          // Tab
//   [\n]          // Newline
//   [\r]          // Carriage return
//   [\x00]        // Character 0 (hex)
//   [\x20]        // Space character
//   [\x30-\x39]   // Digits done another different way
//   [\x20ac]      // 16 bit character
//   [\x01f7e6]    // 21 bit character - Emoji blue square
//   [\\]          // Backslash character
//   [\]]          // Closing square bracket
//   [\-]          // Hyphen character
//   [\^]          // Caret character
//
// 23/04/2020 - Amended to include 5 digit hex values
//
// -------------------------------------------------------------------------------------

%title        = "LaCoGen Grammar - Set literal definition";
%version      = "0";
%author       = "Duncan Munro";
%codeprefix   = "LCG_LITERAL";
%start        = <Grammar>;


// ======================================================================
// Predefined Character sets
// ======================================================================

{Printable} = [ -\xff];
{Number}    = [0-9];

// ======================================================================
// Character sets
// ======================================================================

{EscapedControl} = [\\\-] + [\[\]];
{EscapedSingles} = [tnr];
{EscapedAll}     = {EscapedControl} + {EscapedSingles};
{NonEscaped}     = {Printable} - {EscapedControl};
{HexDigit}       = {Number} + [ABCDEFabcdef];

// ======================================================================
// Terminals
// ======================================================================

CharHex         = "\x" {HexDigit} {HexDigit};
CharNonEscaped  = {NonEscaped};
CharEscaped     = "\" {EscapedAll};


// ======================================================================
// Rules
// ======================================================================

<Grammar>       : "[" <LiteralDefs> "]"         # ActLitSetDefs
                | "[]"                          # ActLitSetEmpty
                |                               # ActLitIgnore
```

```
                ;

<LiteralDefs>   : <LiteralDefs> <LiteralDef>    # ActLitIgnore
                | <LiteralDef>                  # ActLitIgnore
                ;

<LiteralDef>    : <Character> "-" <Character>   # ActLitLiteralRange
                | <Character>                   # ActLitLiteralSingle
                ;

<Character>     : CharNonEscaped                # ActLitCharNonEscaped
                | CharHex                       # ActLitCharHex
                | CharEscaped                   # ActLitCharEscaped
                ;
```

Grammar File for Example Program (test_expr)

```
/*-------------------------------------------------------------------------------

  LaCoGen - Test Expression Evaluator - Floating point version

  05/04/2020

-------------------------------------------------------------------------------*/

%title        = "LaCoGen TEST Expression Evaluator (Floating Point)";
%version      = "0";
%author       = "Duncan Munro";
%codeprefix   = "FP";
%start        = <Command>;


// ===========================================================================
// Predefined Character sets
// ===========================================================================

{digits}        = [0-9];
{alpha}         = [A-Za-z];
{alphanumeric}  = {alpha} + {digits};
{wschar}        = [\t\n\r ];


// ===========================================================================
// Terminals
// ===========================================================================

number      = {digits}* "."? {digits}+ (("E"|"e") ("+"|"-")? {digits}+)? ;
id          = {alpha} {alphanumeric}* ;
help        = MIXED("HELP") keyword ;
print       = MIXED("PRINT") | "?" keyword ;
let         = MIXED("LET") keyword ;
whitespace  = {wschar}+ ignore;

// ===========================================================================
// Rules
// ===========================================================================

<Command>   : <PrintCommand>                                # procignore
            | <AssignCommand>                               # procignore
            | <HelpCommand>                                 # procignore
            ;

<PrintCommand>  : print <Expression>                        # procprint
                ;

<AssignCommand> : let id "=" <Expression>                   # proclet2
                | id "=" <Expression>                       # proclet1
                ;

<HelpCommand>   : help                                      # prochelp
                ;

<Expression>    : <Expression> "+" <AddOp>                  # procadd
                | <Expression> "-" <AddOp>                  # procsub
                | <AddOp>                                   # proccopy
                ;

<AddOp>         : <AddOp> "*" <MulOp>                       # procmul
                | <AddOp> "/" <MulOp>                       # procdiv
                | <MulOp>                                   # proccopy
                ;

<MulOp>         : <PowerOp> "^" <MulOp>                     # procpower     // Right
associative
                | <PowerOp>                                 # proccopy
                ;
```

```
<PowerOp>       : "(" <Expression> ")"                      # procbrackets
                | id "(" <Expression> ")"                   # procfunc1
                | id "(" <Expression> "," <Expression> ")"  # procfunc2
                | <NumberVal>                               # proccopy
                | id                                        # procid
                ;

<NumberVal>     : "+" number                                # procunaryplus
                | "-" number                                # procunaryminus
                | number                                    # procnumber
                ;
```

Appendix D. Release History

Date	Version	Notes
06-Feb-2020	0.1	Initial release of the library code
31-Jan-2021	1.0	Document completed for release