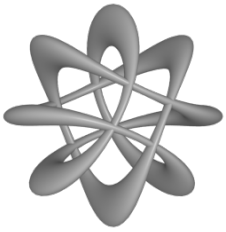
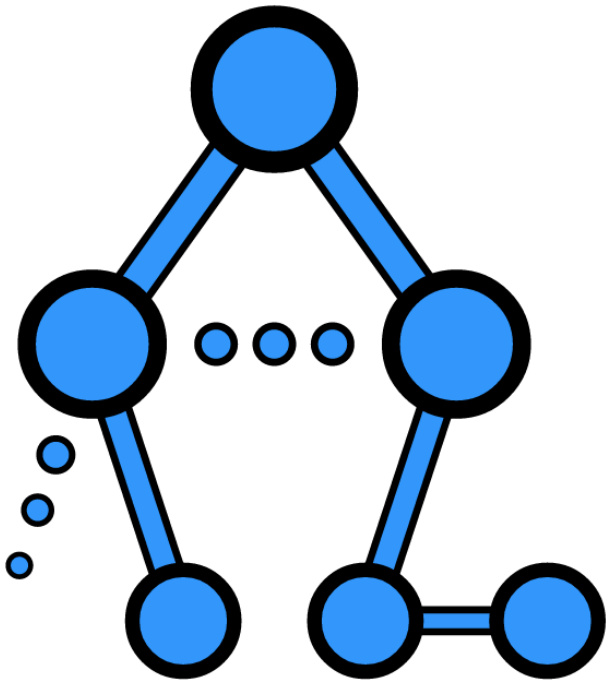


Provided by ExperaSoft



Tunnel Grammar Studio
Software Developer's Manual

This document is provided by ExperaSoft UG (haftungsbeschränkt), Goldgasse 10, 77652 Offenburg, Germany. For more information please visit www.experasoft.com.

Document date 2019/08/26

Contents

Abstract

Tunnel Grammar Studio™ is an Integrated Development Environment (IDE). It is used to develop stand alone[1] Parsing Machines (PM) from given Augmented Backus-Naur Form (ABNF) syntax grammar[2]. Tunnel Grammar Studio handles parsing deterministically for many types of ambiguous grammars. Each generated PM: can be **single** or **multi** threaded; input different text **encodings**; optionally preprocess the input characters and group them into **phrases**[3], for faster parsing and possibly removing language ambiguities; use the **dynamic memory** for in depth parsing (preventing stack overflow problems); emit **syntax errors** for the input supplied with detailed error location, all **expected** at this point tokens and the **current** erroneous token; returns on successful parsing an **explicit concrete syntax tree** (ST). The generated PM source code[4] interface is **object oriented** including the resulting ST. The Tunnel Grammar Studio is doing analysis on the grammars and displays messages for many ambiguities **at compile time**, helping the developer to create deterministic unambiguous grammars that run in **linear time**. The ABNF parser in Tunnel Grammar Studio is designed in itself.

[1] Parsers that does **not** need any external libraries in their run time

[2] Defined in RFC 5234 (Internet Standard 68) and updated by RFC 7405 for string sensitivity

[3] Using a second lexer grammar

[4] Using standard C++98, with Win32API for the optional multi-threading

Introduction

The creation of a capable parser for a given language is time consuming, error prone task. For more complex languages its very hard to keep track of the determinism level at the grammar development stage. Tunnel Grammar Studio™ attempts to solve this problems by generating **object oriented** parsers from supplied ABNF syntax grammars as defined in section 1.4. The input of the generated PM is in bytes, that can be decoded by many different optionally available decoders[5] documented at section 5.3. The decoded input sequence forms Unicode char array that may pass a lexical analysis where one or more characters can be grouped to **phrases**[6], documented at section 1.4.2. The phrases are recognized by the parser grammar as a single token. This method of two phases parsing effectively may parse some ambiguous languages deterministically. During runtime, the PM are emitting events per input **syntax error** discovered, that optionally contain **byte offset** of the error, **Unicode code point offset** or **textual line and line character** offsets. Additionally the syntax error message may contain information of the **current** not recognized token as well as a list of all possible **expected** tokens at the error location. The syntax error events are documented in section 5.6. The result of a successful parsing by a PM is an **explicit concrete syntax tree** that can be iterated as much as need before its destructed. The parsing process uses **dynamic memory** for in depth recursion, and only few function calls depth are made using the dedicated thread stack (DTS). As a consequence, the DTS may be significantly reduced, especially important in server applications. The PM may be run in a **single thread** (executed in **steps** or **till completion**: error/success) or in up to three threads for **multi-threaded** (MT) parallel parsing, where each thread operates on specific part of the **parsing pipeline**[7], documented at section 2, that may bring noticeable speed up for the PM especially for longer inputs. *How to use*, with target the language C++, use cases are at section 5.14. Additionally grammar *examples* are available at 6.

[5]ASCII, ISO 8859-1 (Latin1), WIN 1252, UTF8, UTF16(LE/BE) and UTF32(LE/BE)

[6]Extension to the ABNF grammar

[7]The PM does not spawn threads to do parsing for different ambiguous cases, but splits the parsing in sequential tasks that are run by different threads, effectively creating a pipeline

Chapter 1

Basics

The ability to ‘read‘ is in the core of many computer programs. Reading is connected to finding a meaning of the read information often defined as a string of symbols. Most of the times a single meaning is expected to be found and in case of a computer program this unique meaning is expected to be found **fast**. The process of reading is called parsing, and a string is syntactically ‘correct‘ if it belongs to the given language. A grammar is used to define a language syntax. If the string is not correct, its often important to know where in its symbols is the first error and why the error happened. The process of developing a grammar, especially big and complex, is often hard, additionally the changes of this grammar are even harder, but with a proper tool, a hard task can become much easier. This topics and more are covered and answered by Tunnel Grammar Studio^{TM}.

For clarity this manual uses keywords (”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL”) as defined in RFC 2119[1] by The Internet Engineering Task Force (IETF).

1.1 Parsing

Strictly the parsing is a process of understanding the exact meaning of a string of symbols (called ‘input‘ of ‘tokens‘ later on) conforming a formal grammar. For direct and short example the string of three symbols ’1+2’ and a given formal grammar describing mathematical expressions, the parsing is a process of understanding that this is a summation of two numbers by one digit each, the first is one and the second is two. After this knowledge for the string is collected a mathematical processor could realize the summation and output a solution ’3’. Its important to note that this processor MAY not know how the parsing process was done, only that the input it have to operate on is correct.

1.2 Syntax Tree

The result of a parsing process is often a Syntax Tree (ST). This is a tree, that represents the elements found in the process and their relations. For the example up, the ST could be with a root the sign '+' and two children: '1' and '2'. Given this structure, the mathematical processor could operate on the ST without prior knowledge of how it was constructed - result from a parsing process, generated by an algorithm or loaded from a stream. This devision between a processor and an input generator brings a lot of flexibility to the developers (clear separation of tasks, easy to exchange parts from the whole and so on). There are several types of ST based on different criteria. From the representation perspective:

- Concrete ST (CST) - directly constructed to match fully the grammar structure. Tunnel Grammar Studio™ generates this type of trees.
- Abstract ST (AST) - containing some, not all, predefined level of detail

From the availability perspective:

- Explicit - the tree is fully constructed and available to be traversed multiple times. This gives ability to perform more complex analysis, without the usage of additional structures. Tunnel Grammar Studio™ generates this type of trees.
- Implicit - the tree is never constructed, but only the information for its construction is available, usually in steps. This limits the tree usage to only one time.

Additionally the access to a ST may be:

- Functional Paradigm - the access to ST elements is by calling global functions that receive and return basic primitive types of the used programing language.
- Object Oriented Paradigm (OOP) - the tree elements are represented by objects that contain the data relevant to them including references to another objects. Tunnel Grammar Studio™ generates this type of trees.

The construction of a syntax tree can be in derived in two general ways:

- Left most derivation - the tree is constructed as following the more intuitive way, first the parent is created then the left children of it will be constructed then the right. Tunnel Grammar Studio™ generates this type of trees.
- Right most derivation - first the right children are constructed then the left, then the parent will be created and will add the children.

As a summary Tunnel Grammar Studio™ can produce parsers that process the input from **Left to right** and generate **Left most derivation** (LL) syntax trees that are explicit, concrete and object oriented. The determinism can be extended arbitrary resulting in parsers that recognize LL(*) grammars with possible backtracking for LL(k>1).

1.3 Grammar

A set of rules for strings in a formal language[1] that describe how to form valid expressions according to the language syntax using an alphabet[2]. The meaning of the strings (for what they can be used or are they valid in a given context), however, is not defined into the grammar, only the language syntax is. There is many notation techniques that describe grammars as the focus is on Augmented Backus-Naur Form (ABNF).

1.3.1 Context-Free

These are grammars that their rules can be applied regardless of the previous parsed tokens and location in the input. The context-free grammars are of a great practical interest and many programming languages and Internet protocols are defined by such grammars.

Tunnel Grammar Studio^{TM} produces parsers from context-free grammars.

1.3.2 Ambiguity

One grammar is ambiguous if an input exists that can produce more then one ST. A simple example is 1.3 - a grammar that has two equivalent rules. Every input that is recognized by one of them will be and from the other. Because it can't be answered uniquely is input of '0' the rule 'A' or 'B' the grammar is said to be ambiguous. This ambiguity may be not only between rules, but can be caused from internal rules structure.

Example 1.1: Ambiguous grammar

```
A = "0"
B = "0"
```

1.3.3 Determinism

One grammar is deterministic if at any stage of its recognition for every symbol there is at most one action to make as in example 1.3. As a consequence every deterministic grammar is unambiguous. Its important to note that if one grammar is not deterministic this does not mean that its ambiguous. This is because the non-determinism can be resolved before the input is fully processed, and the possible ST is only one per input.

Example 1.2: Deterministic grammar

```
A = "0" / "1"
B = "2"
```

[1] Set of words over the alphabet

[2] All symbols/tokens that are valid in the language

Example 1.3: Non-deterministic unambiguous grammar

```
A = "0" "1" / "0" "2"
```

1.3.4 Recursion

More complex grammars are containing rules that are referencing each other. The references can be of two types:

- Tree - one rule 'A' may reference another rule 'B', but rule 'B' does not reference 'A' directly or indirectly. See example 1.4. The ST generated for these grammars are with maximum deep equal to the reference tree height.
- Graph - rule 'A' references 'B' and 'B' may directly reference 'A' or indirectly through other rule 'C'. See example 1.5. This ST generated are with theoretically unbounded maximum deep that depends from the input.

Example 1.4: Grammar with references as a tree

```
A = "0" / "1" / B / C
B = "2"
B = "3"
```

Example 1.5: Grammar with references as a graph

```
A = "0" B
B = "1" A / C
C = "2" A / "3"
```

Left Recursion

If one rule is referencing itself before any input token is processed, this is called a direct left recursion as in example 1.6 for a language that recognizes input of '0' followed by zero or more '1'. Regardless of the method used for parsing, the resulted ST will be a deep representation of rule 'A' into 'A' ... with a depth depending from the input.

Example 1.6: Left recursive grammar

```
A = "0" / A "1"
```

Right Recursion

If a grammar rule finishes with a reference to another rule, then a right recursion exists as it is in example 1.7.

Example 1.7: Right recursive grammar

```
A = "0" A / "1"
```

Tunnel Grammar Studio™ accepts any recursive grammars with exception of left recursion.

1.4 Augmented Backus-Naur Form

This notation technique for context-free grammars is defined by RFC 5234[2] (Internet Standard 68) and updated by RFC 7405[3] for string sensitivity. Both standards are operational in Tunnel Grammar Studio™ for defining the Lexer and Parser grammars with special cases, extensions and exceptions handled as define in this section.

1.4.1 Syntax

A valid document with ABNF syntax is consisting from a mixture of comments and rules. Each comment is starting with the symbol ';' (0x3B or decimal 59) and continues till the line termination symbols CRLF (two as 0xD 0xA or decimal 13 10) that is the Internet standard new line. Each grammar rule is consisting of elements, some elements may be containing another inside itself. With exception of the optional group every element MAY be repeated in a range between a minimum (a number grater or equal to zero) and a maximum (a number grater then zero, or an infinity). The strings are surrounded by double quotes, and as of RFC 7405[3] can be sensitive or insensitive. For the full specification please refer to RFC 5234[2], and for summary of the used definitions from Tunnel Grammar Studio™ see bellow:

- `rulename = ALPHA *(ALPHA / DIGIT / "-")`

  This is a string of symbols that identifies the rule name for its definition and for its references in other rules. Note that underscore is not permitted. For how this name participate in the generated code please see in the target language section. Comment or white space (CWS) is not permitted in this definition.

- `rule = alternation CRLF`

  Each rule is consisting of sub alternatives and ends in a line terminator. Additionally one rule may be altered by adding additional sub alternatives by using token "/=" instead of "=". Each rule definition must start at the beginning of a line. CWS is permitted around the definitions.

- `alternation = concatenation *("/" concatenation)`

  Each alternative element consist from sub concatenations of elements each split by symbol "/". CWS is permitted around the definitions.

- `concatenation = 1*repetition`

  Each concatenation is a list of repetitions. CWS is permitted between the definitions.

- `repetition = [1*DIGIT / (*DIGIT "*" *DIGIT)] element`

  Each element may be repeated with a range of minimum and maximum, where minimum can be any number, if omitted zero is assumed by default, and the maximum is a number bigger or equal then the minimum, if omitted infinity is assumed by default. For example 2*5 is 'from two to 5 times', and 1* is 'at least one time' or 'from 1 to infinity', *5 donates 'no more then 5' or 'from zero to 5 times', and a star symbol alone means 'any count' or 'from zero to infinity'. The standard does not permit CWS between the element and the repetition group, and also between the star delimiter and the minimum and maximum numbers, but Tunnel Grammar Studio™ permits. This is made purely for easy to format grammars. Additionally the standard allows zero repetitions of elements, that is considered from us for a not well formed grammar element and is not permitted in Tunnel Grammar Studio™.

- `element = rulename / group / option / char-val / num-val / prose-val / symbol-val / phrase-val`

  Each element may be one of the sub referenced types. The elements of type symbol-val and phrase-val are not defined by the standard, but recognized by Tunnel Grammar Studio because of the need to have longer Lexer tokens then single symbol, and comfort of the user to write single symbol ranges, more detailed definitions of this element types are later on. CWS is permitted between the definitions.

- `group = "(" alternation ")"`

  To create sub alternatives a group element can be used. The repetition for a group element is set to 1*1, and its not allowed to specify it or any other repetition before the element. CWS is permitted between the definitions.

- `option = "[" alternation "]"`

  To create elements group that MAY be existing into the input stream, this optional group is used. The repetition for a option element is set to 0*1, and its not allowed to specify it or any other repetition before the element. CWS is permitted between the definitions.

- `char-val = ["%s" / "%i")] %x22 *(%x20-21 / %x23-7F) %x22`

  The element of this type declares a string of characters surrounded by double quotes. The double quote is not allowed inside the string. No escaping for the double quotes is possible by the standard, and if one needs this symbol then one MUST use the num-val definitions instead. Is the string going to be matched case sensitively (%s prefix) or not (%i prefix), is defined before the string, if omitted, **insensitive** is used by default as defined by the standard. CWS is allowed before and after the element..

- `num-val = "%" (bin-val / dec-val / hex-val)`

This is a definition of string formated by binary, decimal or hexadecimal numbers. CWS is allowed before and after the element.

- `bin-val = "b" 1*BIT [ 1*("." 1*BIT) / ("-" 1*BIT) ]`

  Binary representation of a string. The three types of defining this elements are: a) %b1000001 that donates the decimal %d65 and hexadecimal %x41 with ASCII symbol 'A'. b) %b1000001-1000003 that donates a range of an ASCII symbol ['A'..'C']. c) %b1000001.1000002.1000003 that donates concatenation of 3 chars 'ABC'. The valid value is set in range [0..0x10FFFF] to cover all Unicode symbols. CWS is allowed after the element.

- `dec-val = "d" 1*DIGIT [ 1*("." 1*DIGIT) / ("-" 1*DIGIT) ]`

  A decimal representation of a string. The three types of defining this elements are: a) %d65 that donates the binary %b1000001 and hexadecimal %x41 with ASCII symbol 'A'. b) %d65-67 that donates a range of an ASCII symbol ['A'..'C']. c) %d65.66.67 that donates concatenation of 3 chars 'ABC'. The valid values are set in range [0..0x10FFFF] to cover all Unicode symbols. CWS is allowed after the element.

- `hex-val = "x" 1*HEXDIG [ 1*("." 1*HEXDIG) / ("-" 1*HEXDIG) ]`

  A hexadecimal representation of a string. The three types of defining this elements are: a) %x41 that donates the binary %b1000001 and decimal %d65 with ASCII symbol 'A'. b) %x41-43 that donates a range of an ASCII symbol ['A'..'C']. c) %x41.42.43 that donates concatenation of 3 chars 'ABC'. CWS is allowed after the element. The valid value is set in range [0..0x10FFFF] to cover all Unicode symbols..

- `prose-val = "<" *(%x20-3D / %x3F-7F) ">"`

  Representation of a string that does not contain symbol "<". CWS is allowed before or after the element.

- `symbol-val = %x27 (%x20-26 / %28-7F) %x27["-" %x27 (%x20-26 / %28-7F) %x27]`

  This is an **extension** of the standard, for defining direct symbol definition surrounded by single quote plus optional range. This is a comfort for the grammar developer, because the ABNF syntax permits ranges defined by numbers (binary, decimal or hexadecimal), but that is hard to read code (as for example is this symbol-val definition written up, that reads as 'single quote, then a space or visible symbol (without the single quote), followed by a single quote', then optionally 'dash' followed by a space or visible symbol (without the single quote) surrounded with a single quotes). By making the syntax extended by this element type one may simply write: `name = '"' ('A'-Z' / 'a'-'z') *('A'-Z' / 'a'-'z' / '0'-'9') '"'` that is far easier to read and work with then: `name = %x22 (%x41-5A / %x61-7A) *(%x41-5A / %x61-7A / %x30-39) %x22` that are essentially the same as a functionality. CWS is allowed before and after the element and after the sensitivity prefix.

- `phrase-val = "{" %x22 rulename %x22 ["," (char-val / num-val / prose-val)] "}"`

  This is an **extension** of the standard, because of the need an element to match a longer then single symbol token sent by the Lexer documented in a section 1.4.2. If a simple match of a Lexer token is required, one may define `A = { "word" }`, that will match any "word" type of token send by the Lexer. If one wants to match specific content of the token, one may write `A = { "word", %s "int" }` to match case sensitively the 'word' token type with a symbol string 'int' case sensitively. Refer to sections 2.3 and 2.4 for the benefits this element gives in practice. CWS is allowed around the delimiters.

- `ALPHA = %x41-5A / %x61-7A`

  That are chars a-z and A-Z inclusive defined in hexadecimal form. Using the symbol-val extension this could be defined as `ALPHA = 'A'-'Z' / 'a'-'Z'`.

- `BIT = "0" / "1"`

  Bit can be symbol zero or one.

- `DIGIT = %x30-39`

  A digit is any number form zero to nine inclusive. Using the symbol-val extension this could be defined as `DIGIT = '0'-'9'`

- `HEXDIGIT = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"`

  Any digit form zero to nine plus any lower and capital letter from 'A' till 'F' inclusive. Note: the default char-val string is case insensitive and "A" matches "a" and "A", but if defined as a number %d65 only 'A' is matched.

- `CRLF = %x13 %x10`

  Internet standard line terminator - Carriage return (CR) followed by a line feed (LF).

An iterative solution to the recursive grammar in example 1.6 would be to refactor it as in example 1.8. However the syntax tree will be different, regardless that the language produced by the both grammars is the same.

Example 1.8: Simple repetition grammar

```
A = "0" *"1"
```

1.4.2 Phrases

Tunnel Grammar Studio™ is designed to use 2 grammars for producing a parsing machine (PM). Those are Lexer and Parser grammars. The first produces tokens from the input symbols, and a token MAY be longer then a single symbol. Because the ABNF standard defines only direct input by single symbols we have made an extension to the

ABNF grammar to be able to match this longer then single symbol tokens produced from the Lexer. This effectively makes 2 levels of the input recognition (lexing and parsing) described more in details at section 2. This simplifies the grammar definitions, speeds up the process of recognition and MAY remove some undesired ambiguities in some grammars.

1.4.3 End Of File

Tunnel Grammar Studio™ have spacial token send by the Lexer when the input has finished. It can be matched in the Parser grammar with {$EOF}.

1.4.4 Epsilon Paths

The ABNF syntax permits shippable elements by defining zero for a minimum repetition, referring to the parser SAP element. This effectively makes it possible to have a rule that may be successfully recognized without any token be used from the input, as in example 1.9.

Example 1.9: Epsilon Rule

```
A = 0*1 'x'
```

Because a ‘rule‘ is defined to be an alternation of concatenations, the same for a ‘group‘ and ‘option‘ ABNF elements, more then one epsilon path (EP) may exists and as a consequence more then one syntax tree can be constructed in this places. To avoid this ambiguity, that does not result from the input directly, but from the grammar, Tunnel Grammar Studio™ chooses one of all syntax trees possible, with optimized calculations for faster to construct and smaller in memory syntax sub tree. For the grammar in example 1.10 there is 2 EP possible to construct the rule 'A': one is to use the first alternative and have zero of 'x' and the second is to use the second alternative by having zero of the group 'y' 'z'. In this case the first alternative is faster to construct and additionally smaller in the memory and will be automatically chosen.

Example 1.10: Epsilon rule in a parser grammar

```
A = 0*1 'x' / ['y' 'z']
```

The Tunnel Grammar Studio does report with a warning the places where multiple EP exists in the grammar, and all this places are parsed linearly (i.e. O(1)) with the precalculated path. It is also possible to have an infinite EP, that also are O(1) parsed by every PM, as it is in example 1.11 where it is defined an infinity of an infinity of 'x'. This is automatically resolved by Tunnel Grammar Studio™for input 'y'. However, there exists multiple representations of input 'xxy', that will be iterated each in turn. This grammar is not LL(1) because of the infinity of infinity of 'x' collision, and it will be reported by

Tunnel Grammar Studioincluding the epsilon paths location. It is RECOMMENDED not to have multiple paths in the parser grammar at all. The lexer grammar MAY have any kind of epsilon paths.

Example 1.11: Epsilon rule in a parser grammar

```
A = *(*'x') 'y'
```

1.5 Epsilon repetitions

By definition an ABNF grammar element have a range for its repetition. If this element have an epsilon path inside itself as in example 1.12, then becomes ambiguous, where in the repetition to place this empty recognized rules. Tunnel Grammar Studio™ resolves internally this cases and parses them linearly (i.e. O(1)) regardless of the placing the empty rules in the repetition list. For the example, the input of 'xxy' will have a single representation of the syntax tree with 2 times 'B' with one x inside each, 3 epsilon paths of 'B', and 'y' in the end. The other combinations as for example is to have EP of 'B' one time, then 1 time 'B' with 'x' inside, then 2 times EP of 'B' and then one time 'B' with 'x' inside, ending with 'y'. The total combinations possible are $C\binom{5}{2} = 10$, but only one will be tested from the generated PM, effectively removing the ambiguity caused by the range repetition of a EP element, in this case the rule 'B'. It is RECOMMENDED not to have epsilon paths in the parser grammar at all. The lexer grammar MAY have any kind of epsilon paths.

Example 1.12: Epsilon repetition

```
A = 2*5 B 'y'
B = 0*1 'x'
```

One way to remove this ambiguity is to transform the grammar from example 1.12 into the grammar in example 1.13.

Example 1.13: Epsilon repetition refactored

```
A = 2*5 B 'y'
B = 'x'
```

Chapter 2

String Analyzing Pipeline

The string recognition process in Tunnel Grammar Studio™ is divided in three main parts: Lexing, Parsing and Syntax Tree Construction (called for short Building). A PM is created by 2 individual grammars one for the Lexer and one for the Parser. Organizing the process as a pipeline, gives a great possibility of multi-threaded support already available in Tunnel Grammar Studio™ and documented in section 3.6. The String Analyzing Pipeline (SAP) elements are:

2.1 Source

This module supplies the raw bytes of the input. It can be any kind of stream: file, memory, network socket, be generated in real-time or other. The Tunnel Grammar Studio is generating a pure class that can be extended to any type of source, and generates **memory source** (reading from a byte array) in every PM.

2.2 Decoding

There is default decoder available in each PM for ASCII character encoding and optionally available decoders for Win1252, ISO-8859-1 (Latin1), UTF-8, UTF-16 LE/BE and UTF-32 LE/BE. This element is organizing (decoding) the raw bytes into symbols.

2.3 Lexer

Each received token from the previous decoder pipeline element (PE) is used as a symbol. The Lexer grammar is used to recognize phrases from the list of symbols. If a phrase is recognized, it is transmitted to the parser as a single **phrase** token. If no phrase is recognized a single symbol is transmitted as a **symbol** token and the remaining symbols are again tested with the grammar rules. The Lexer grammar is with ABNF syntax, with the following exceptions:

- Elements may be repeated only with minimum equal to zero or one and maximum equal to one or infinity (this means that allowed are: '*' as zero to infinity; 1* as one to infinity, 0*1 as zero one time and 1*1 that is the default value if no repeater is present before grammar element)
- No phrase-val is allowed, because there is element before the Lexer that can send such a token.
- No pseudo-recursion is allowed. Meaning that if A references B, it cant reference A backward directly or indirectly.
- The system token {$EOF} can't be used.

The grammar SHOULD recognize words and simple phrases only, and SHOULD be kept simple and short. For a word and number splitter lexer grammar see example 2.1

Example 2.1: Simple word/number splitting Lexer grammar

```
WORD = 1* ('a'-'z' / 'A' - 'Z')
NUMBER = 1* '0'-'9'
```

The Lexer grammar rules may collide, in such a case the rule defined later in the document have a precedence. For maximum speed of recognition the lexer grammar is expanded into a Determined Final Automat (DFA). This expansion MAY be significant in size if the grammar is not kept short or the ambiguities inside are too much.

2.4 Parser

The input of the SAP Parser (SAPP) Element is a token from the Lexer. The full ABNF syntax is supported as defined in section 1.4. For maximum performance many possible states of the parser are explicitly created, which MAY result in significant target code size, for this reason it is RECOMMENDED to keep the grammar short and clear of collisions. Tunnel Grammar Studio™ is implementing many analysis algorithms with the goal to catch the runtime collisions at compile time, so the grammar developer can design more faster running Parser grammars. If the supplied grammar is LL(k>1) then the parser MAY backtrack. If the grammar is LL(1) then the Parser PE is expected to run in linear to the tokens speed - O(n). For each operation the parser is performing commands for building the syntax tree are send to the next PE.

2.5 Optimizer

This element is buffering the commands sent from the Parser PE. In case that a backtrack is required if the Optimizer have buffered commands, this commands will be erased and never transmitted to the Builder. This MAY provide significant speed up in the parsing process especially (but not only) when backtrack is realized.

2.6 Builder

The received commands from the Parser (thru the Optimizer) are used to construct the final ST, that matches the Parser grammar as it is. In case of a backtrack, the SAP Builder (SAPB) element adapts the tree to match the current state of the Parser.

2.7 Glue Code

After the ST is complete the client code can use it and free it when no more needed. Glue code is OPTIONAL, but RECOMMENDED. This SHOULD be a code, written by the client, that connects the automatically generated syntax tree source code structures with the actual client program. For examples see section 6. This level of abstraction brings a lot of benefits in the development process, because changes to the Parser grammar MAY require only changes to the glue code, not to the whole client program.

Chapter 3

Parsing Machine

The architecture of the generated PM is as a separate module inside the client program, that requires only standard libraries of the target language to compile i.e. no Dynamic Link Libraries or third party software is required. The design of the PM is:

3.1 Online

The generated PM processes as much as input is available. When no more input is available the machine pauses. When more input becomes available the client code MUST signal this to the machine so it can continue its processes.

3.2 Memory model

The PM uses fixed memory blocks (FMB) for its internal state for languages that provide memory access with pointers (currently C++).

3.3 Run time

The machine may be run till completion: more input is need, till and error is discovered in the input or till successful recognition. Additionally PM can run on small steps as possible - this allows many single thread (ST) PM to execute in one dedicated thread in turns.

3.4 Code diversity

Each time the PM is generated from Tunnel Grammar Studio the parser SAP element code is randomized. This effectively makes the reverse engineering more difficult of the

parser grammar after its compiled into an executable.

3.5 Dynamic stack

The generated Parser SAP element does not use the dedicated thread stack (DTS) for in deep grammar execution, but creates its own dynamic stack. This ensures that at runtime the client program is not going to collapse from missing stack space, because of grammar recursions. Some function calls deep in the PM code MAY occur, but the depth is independent from the grammar recursions.

3.6 Threading

One PM may be compiled to run as single or multi threaded (MT). Currently for MT three threads are started per PM instance. Because of the dynamic stack each PM maintains, each thread that runs the PM can be started with very little DTS (this is valid for multi-threaded and for single threaded client thread). Because the PM instance is called from the client code, and the PM does some system calls (as at least to allocate memory), its not possible to calculate in advance the maximum DTS size needed. However, because no in deep recursion is going to happen on DTS, a client program MAY shrink the parsing threads stack significantly. This brings high scalability (because a default TDS is usually in megabytes) in case of the parsing is used in a server application.

Chapter 4

Studio

Tunnel Grammar Studio™ has a compiler build in for ABNF to a Target source code (TSC), currently C++. There are two main tabs in the Graphical User Interface (GUI) that contain code editors for the Lexer and Parser grammars with ABNF syntax with exceptions, spacial cases and extensions defined in sections 1.4 and 2.

4.1 System Requirements

The Tunnel Grammar Studio runs currently in Windows XP, 7 and 10. The GUI uses OpenGL with minimum supported version 2.0 with minimum of 128MB video memory. At runtime, depending from the grammar the video memory requirements MAY grow. The runtime memory need is minimum 32 MB, but at compile time the memory required MAY grow significantly based on the supplied grammars. Hard disk space required for the moment is less then 10 MB.

4.2 Installation

The Tunnel Grammar Studio™ is a standalone application with a single executable file and an eventually accompanying license file. Every registered user after a successfully completed purchase of a license, may download its files from the website in its user page. Every executable file is accompanied by a hash SHA-256 code on the file bytes.

4.3 Lexer

The first tab in GUI is editor for the Lexer Grammar, that will be compiled as the Lexer SAP element as defined in section 2.3. At compilation time first the syntax errors are checked, then an analysis is made to detect references to missing rules, invalid repetition ranges and rules recognition collisions. All found errors and warnings, plus all

messages are logged into the GUI log list. For the error codes (EC) description refer to section 4.9.

4.4 Parser

The second tab in the GUI is an editor for the Parser Grammar, that will be compiled as the Parser SAP element as defined in section 2.4. At compile time, after the syntax is checked, complex analysis are initiated to detect LL(k>1) collisions. All found errors and warnings plus additional relevant messages are logged into the GUI log list. For the EC description refer to section 4.9.

4.5 View

This tab contains the graphical representation of different graphs and relations of the PM. Currently each lexer and parser grammar can be visualized as an automat. Color options for some parts of the graphics can be found the Main Menu/Studio/Options.

4.6 Debug

The Tunnel Grammar Studio^{TM} have a build in virtual parsing machine (VPM) debug engine. It can be used to step by step process an input of characters and understand better the interaction of the lexer and parser grammars. The fast/slow forward/backward processing executes the VPM in the choosen direction and speed and stops at any error (the parsing have to backtrack) or a success (a valid parse tree was found for the given input).

The debugging can help the development of the grammars significantly, because the developers may try different inputs to find and see the resulting parse tree, and compare it to their expectations.

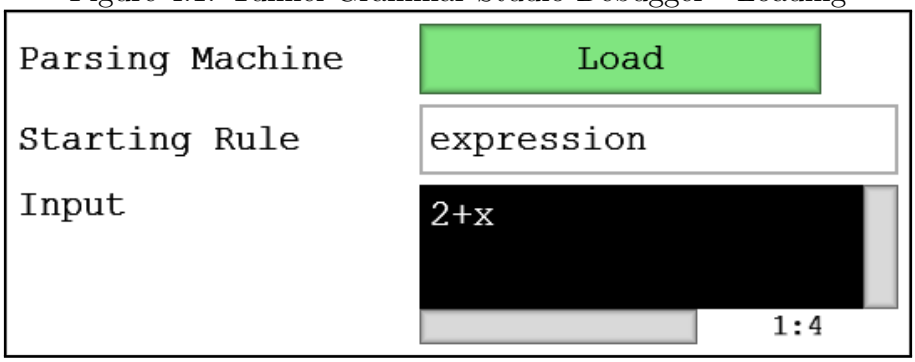
4.6.1 Loading

In loading screen of the debugger, the start parse rule must be supplied and must be one of the rules in the parser grammar. The second piece of information is the input, that will be used from the virtual parsing machine in the time of debugging. The input is represented by UTF-32 chars. The loading screen is shown in figure 4.1.

4.6.2 Debugging

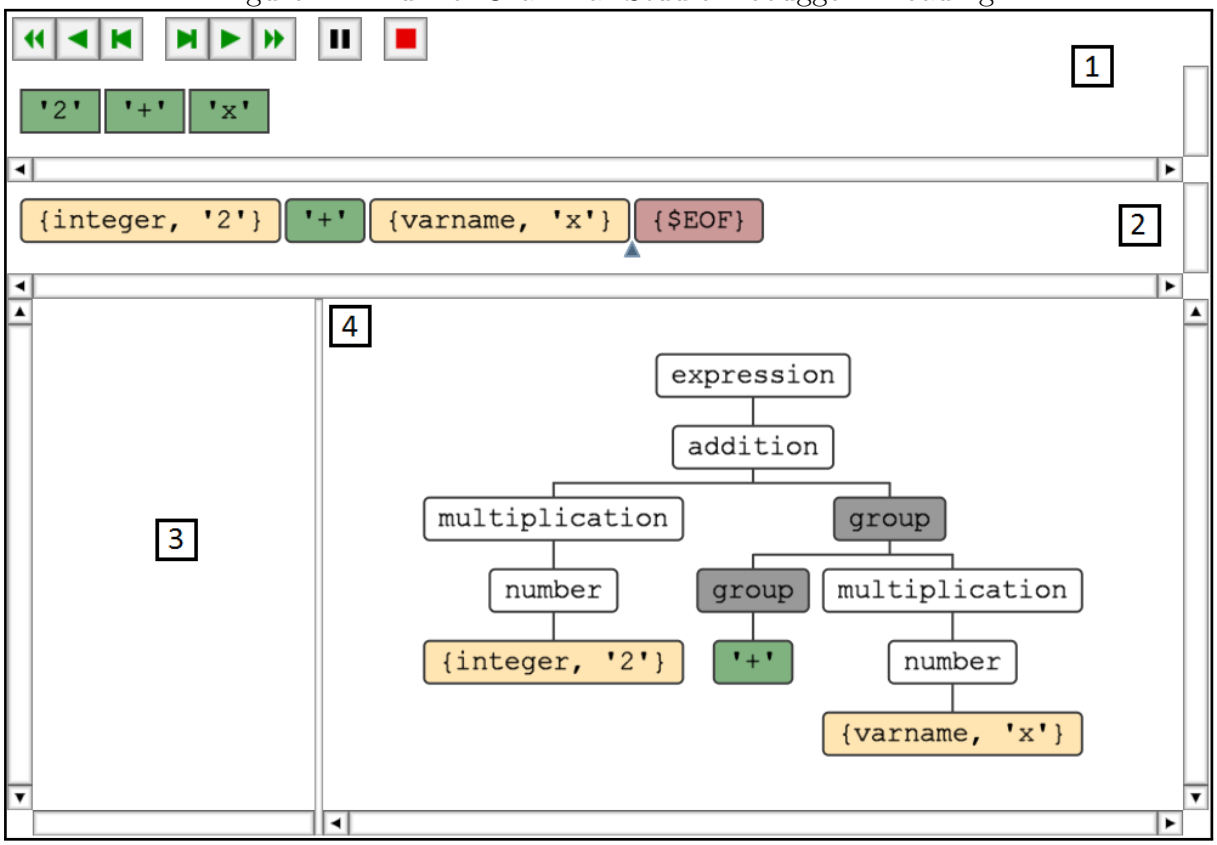
After successful loading the VPM can be runned in steps forward and backward. A result of running the mathematical expression use case (in section) with the input "2+x"

Figure 4.1: Tunnel Grammar Studio Debugger - Loading



gives the first successful parsing as shown in figure 4.2.

Figure 4.2: Tunnel Grammar Studio Debugger - Loading



- Controls - In the top left corner of figure 4.2 are the controls. From left to right the buttons are: fast backwards, slow backwards, single step backwards; forward single step, forward slowly, forward fast; pause if running slow or fast in any direction and stop button that will close the debugger.
- Input View - Contains the single Unicode characters that are read from the stream. Its the area marked with box 1 in figure 4.2.
- Tokens View - Holds the recognized tokens by the lexer grammar from the input characters. Its the area marked with box 2 in figure 4.2.

- Stack View - Lists the current stack items in the process of the VMP runtime. Its the area marked with box 3 in figure 4.2 and its empty because the input was completely consumed and currently the parser is not in any rule.
- Parser Tree View - Displays the current concrete parse tree in the current VMP runtime. It is dynamically updated at every step that affects the tree. All the tokens can be seen in the tree left to right as leafs. Its the area marked with box 4 in figure 4.2.

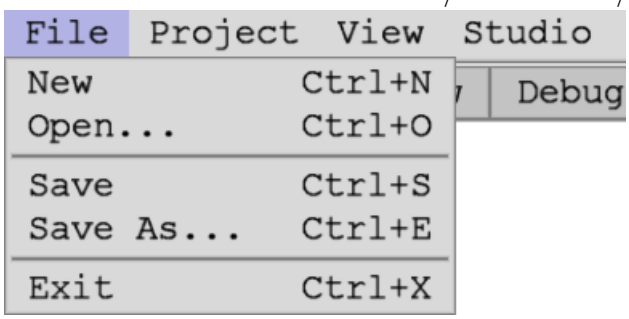
4.7 Main Menu

The Tunnel Grammar Studio^{TM} Main Menu follows this structure:

4.7.1 File

For everything related to the project file as in figure 4.3.

Figure 4.3: Tunnel Grammar Studio/Main Menu/File



- New - clear all project data: lexer and parser editors and the project options.
- Open - open an existing project file.
- Save - save the current project to a file.
- Save As - save the current project to a new file, without changing the current opened file.
- Exit - closes the program after all changes to the project are saved or discarded.

4.7.2 Project

Contains all project options and operations as in figure 4.4.

- Options - all project options organized in tabs.
  - ◇ General - overall setup for the project showed in figure 4.5. The fields are:

Figure 4.4: Tunnel Grammar Studio/Main Menu/Project

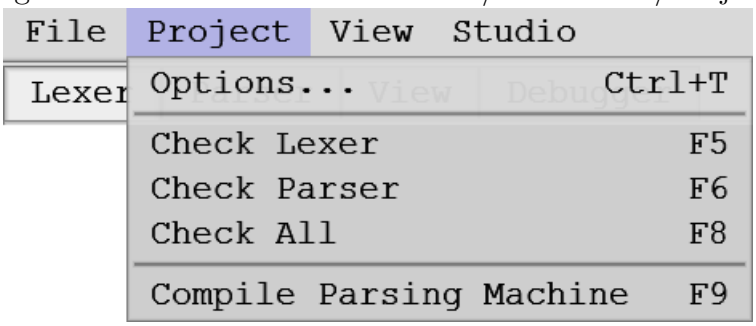
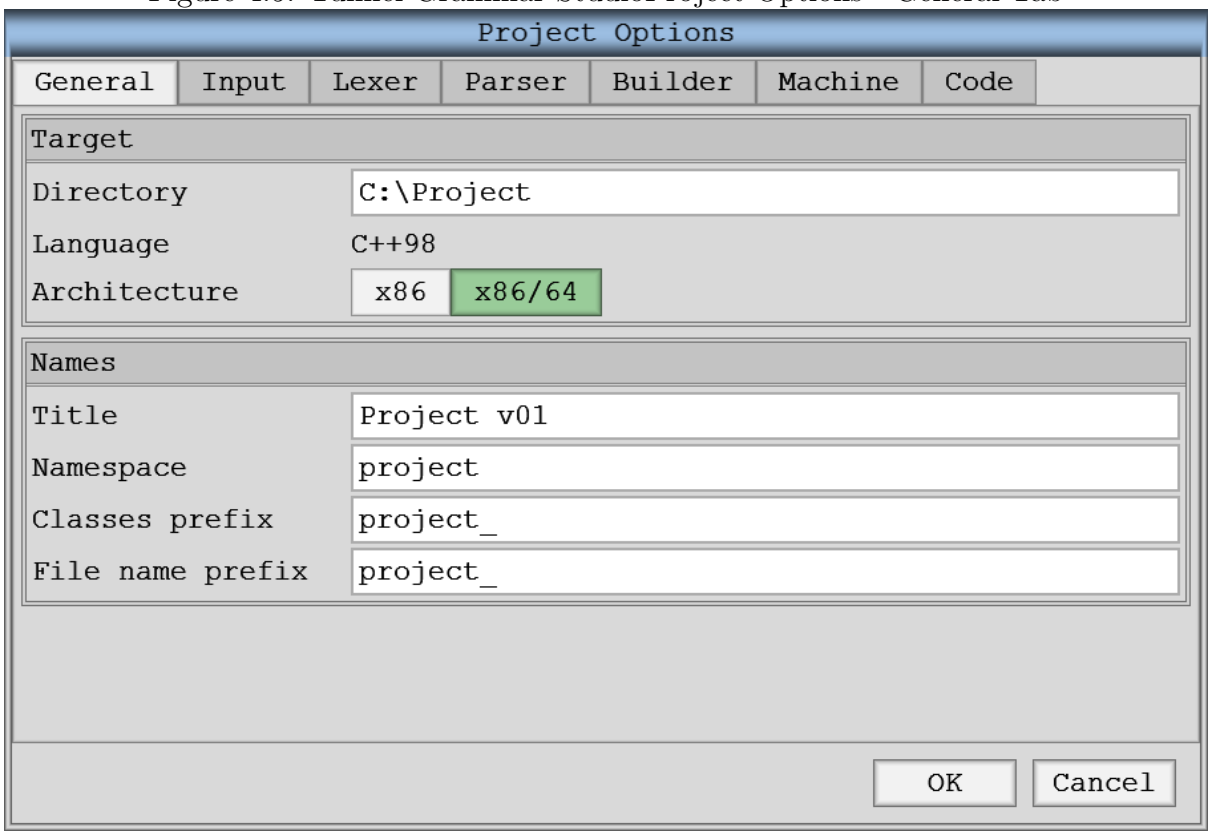


Figure 4.5: Tunnel Grammar StudioProject Options - General Tab

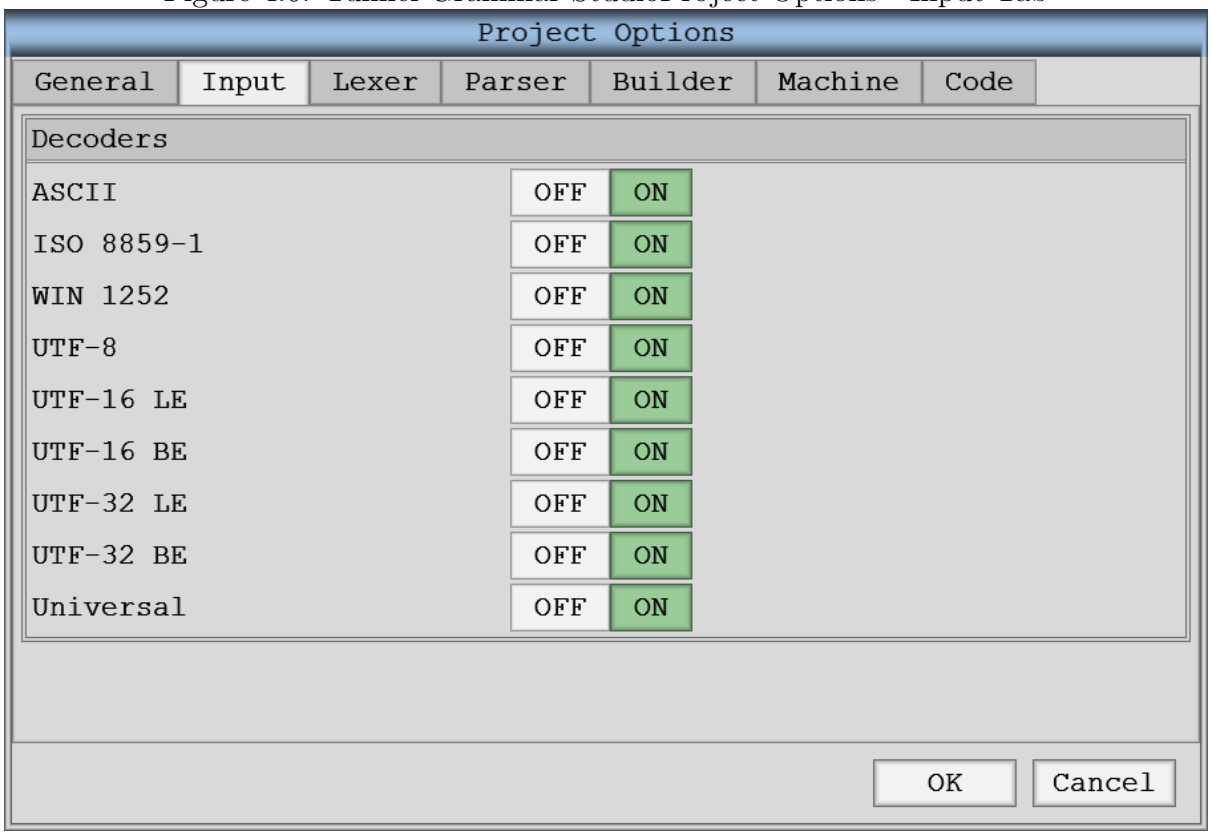


* Directory - this is the directory where the compiled target language files will be placed. Sub folders MAY be created. It is NOT RECOMMENDED to store any other files in this folder, but the compiled, because future version MAY save files with different names and WILL overwrite the existing.
* Title - short descriptive name of the PM (for example 'Language v5').
* Namespace - this is the namespace the source code will be generated to use. If empty, that is NOT RECOMMENDED, no namespace is generated and the code uses the global namespace.
* Classes prefix - for every parser grammar rule will be generated an object, and each of this objects will have name prefix as in this field.
* File Name prefix - every generated file name will be prefixed for easy

recognition of the files when included in a target language project. MAY be empty, then no common prefix of the files will be used.

◇ Input - the switches for having specific input decoders available in the compiled PM, showed in figure 4.6. More details about the decoders are in section 5.3.

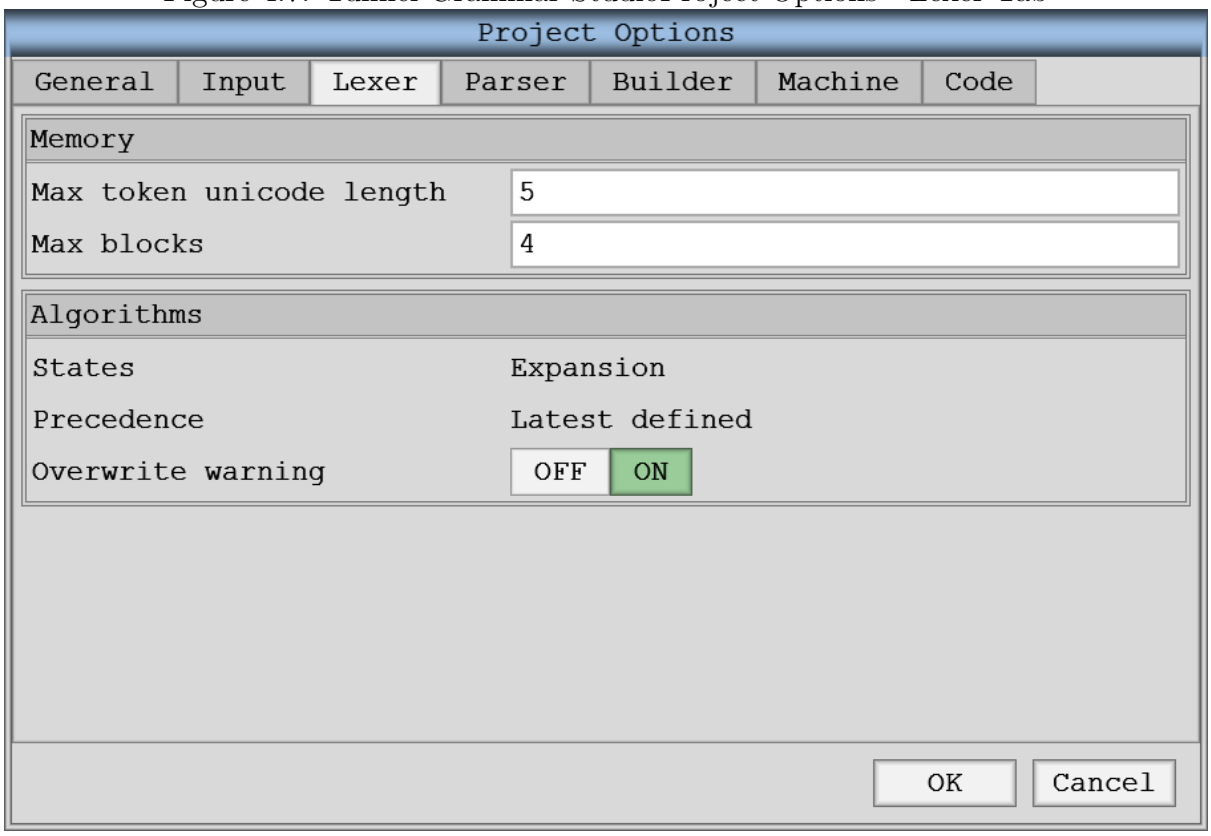
Figure 4.6: Tunnel Grammar StudioProject Options - Input Tab



◇ Lexer - Options related to the generated SAP lexer element showed in figure 4.7. The fields are:

* Max token unicode length - the maximum length of a phrase recognized by the lexer. Valid range is $[1..2^{24}]$.
* Max blocks - maximum memory blocks to use. This is a soft limit any may be extended from the lexer if need (for example longer phrases are recognized).
* States - defines the automat states generation method. Currently only full expansion is available as defined in section 2.3.
* Precedence - defines what lexer rule to be with a higher priority in case of a collision i.e. what token type will be returned if more then one rule can recognize the current input sequence. Currently the *latest defined* is the only option.
* Overwrite warning - should a warning be printed in the log in case of a lexer rules collision. A collision here means that two or more lexer rules

Figure 4.7: Tunnel Grammar StudioProject Options - Lexer Tab



have the same final automata state.

◇ Parser - options related to the SAPP element.

* Max look ahead tokens - defines how much tokens maximum the SAP parser element MAY search for a recognition before terminates with an error. If this field is left empty, an infinity is assumed: LL(*). However, for grammars that are LL(k>1) this MAY result to a worse then linear runtime per input token. The amount of tokens that are kept in the memory is at least this value.

* Max obsolete tokens - this is the amount of tokens that will be kept in the PM internal structures. At any time after this threshold + the -max look ahead tokens- is reached the obsolete tokens MAY be discarded.

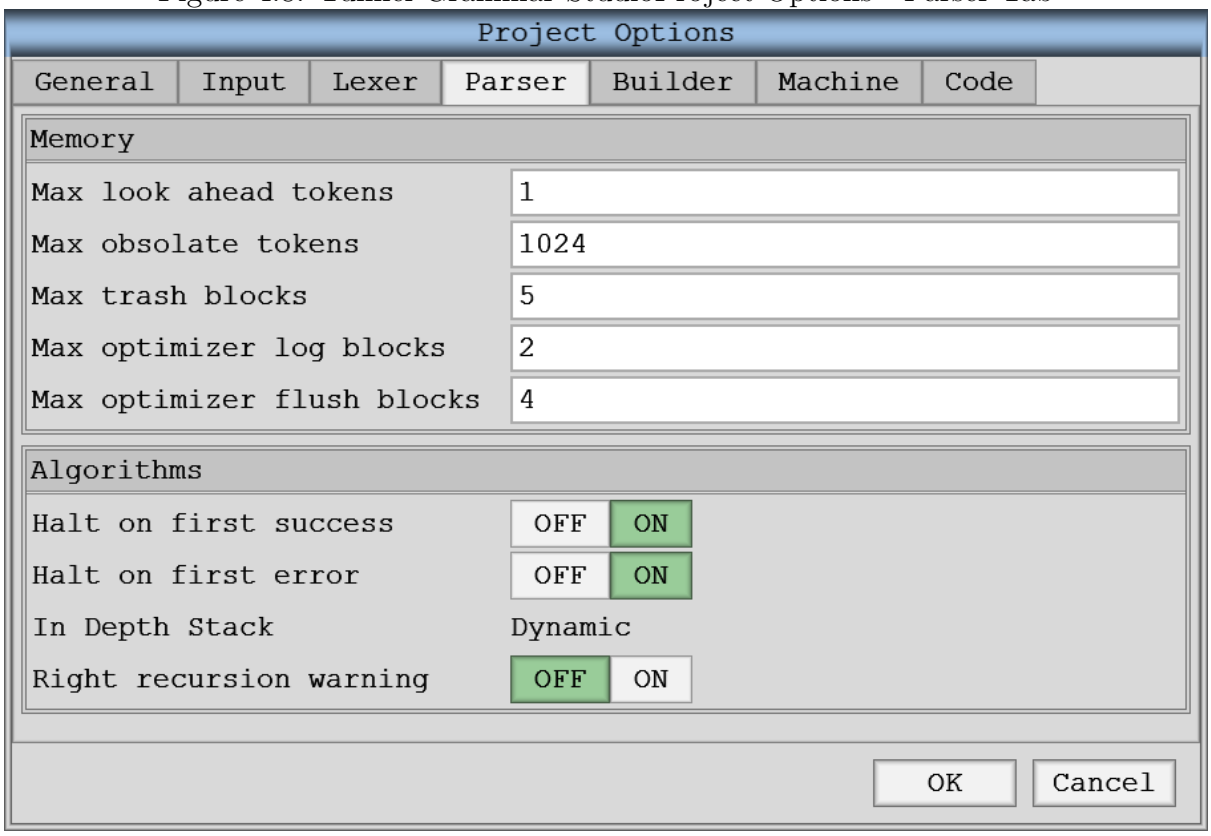
* Max trash blocks - the amount of unused dynamic stack memory blocks that will be freed at ones after so much are available.

* Max optimizer log blocks - the amount of blocks that will be kept available for an optimization.

* Max optimizer flush blocks - the RECOMMENDED amount of ready blocks that will be send to the SAPB.

* Halt on first success - if turned 'on' then the parser will stop the parsing after the first successful input recognition. If turned 'off' the parser will

Figure 4.8: Tunnel Grammar StudioProject Options - Parser Tab



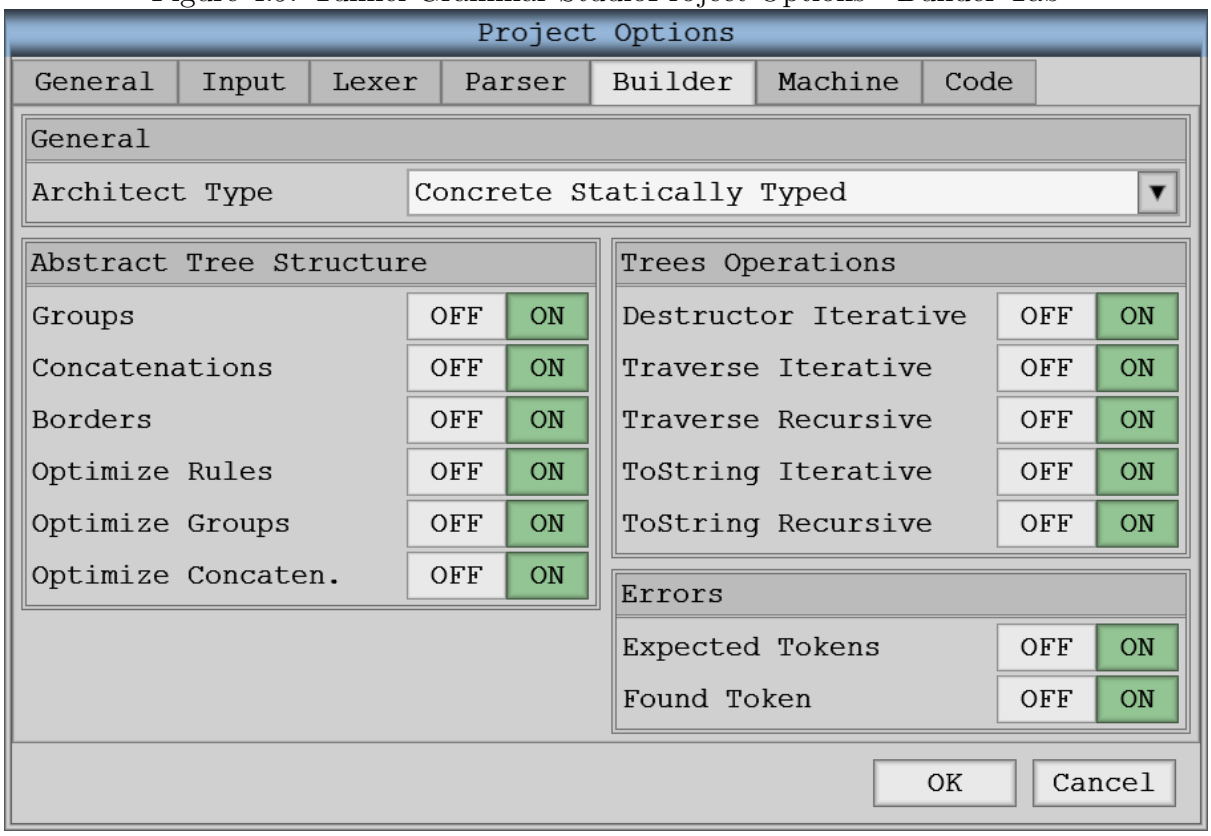
signal the success and backtrack to search for more successful combinations.

- Halt on first error - if turned 'on' the parser will stop on the first error found. If turned 'off' the parser will signal the errors found and will backtrack to search for a success combination.
- Right recursion warning - if one or more rules are part of a right recursion a warning will be printed.

◇ Builder - options related to the SAPB element.

- Architect Type - there are 5 architect types available.
  - None - the architect is missing, and no tree information will be generated. The parser with this architect type will only check for a valid input.
  - Concrete Statically Typed - this architect will construct a concrete statically typed syntax tree. This tree have a separate classes for each rule, group, alternative and concatenation from the whole grammar. Each repetition is represented by an array of elements, and if the minimum and maximum repetitions are not the same from a template list class for the repetable ABNF element.
  - Concrete Visitor - an architect that is only receiving the commands

Figure 4.9: Tunnel Grammar StudioProject Options - Builder Tab



for constructing the concrete syntax tree as a visitor pattern. By using this architect type, one can construct a custom concrete tree.

- Abstract Dynamically Typed - this architect constructs a dynamically typed abstract syntax tree. This tree have a generic class of 'Node' that represents every rule, group and alternative. Each concatenation is represented by a sub node linked list. This tree suports runtime automatic optimization of nodes with one child, thus many different abstractions of the tree are possible.
- Abstract Visitor - an architect that is only receiving the commands for constructing the abstract syntax tree as a visitor pattern. By using this architect type, one can construct a custom abstract tree.

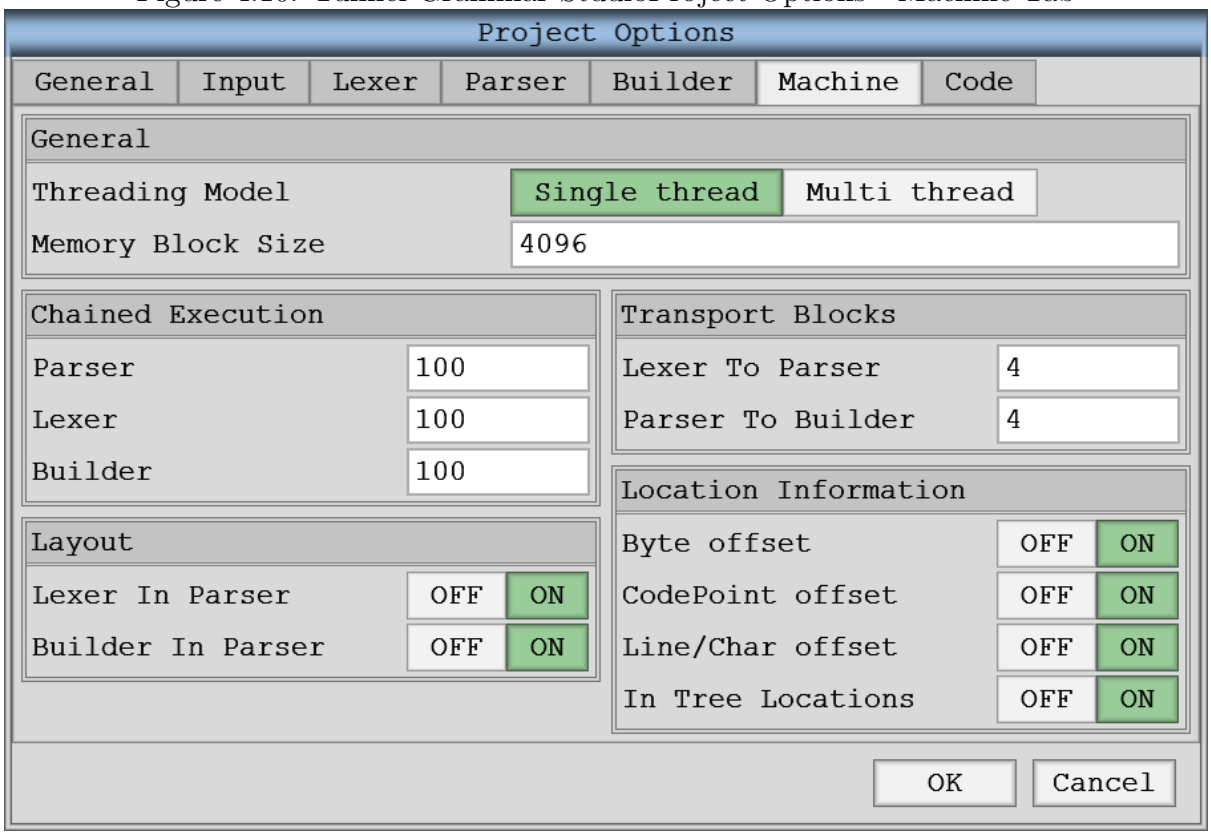
* Destructor with dynamic stack - when the PM is destructed, it destructs any ST available. If this option is turned 'on' the dynamic memory is used to destruct the syntax tree. If turned 'off' that is NOT RECOMMENDED the thread dedicated stack is used for in depth tree elements destruction.
* Traverse Iterative - this option, give to the generated parser classes and functions to iterate the constructed tree (concrete or abstract) in an iterative manner (dynamic memory is used for in depth traversal of the elements, its expected to be slower then the recursive, but stack overflow is not likely). A visitor pattern is used.

- Traverse Recursive - this option, give to the generated parser classes and functions to iterate the constructed tree (concrete or abstract) in a recursive manner (the thread dedicated stack/call stack is used for in depth traversal, its expected to be faster then the iterative traversal, but for very depth trees a stack overflow MAY occure easiliy). A visitor pattern is used.
- ToString Iterative - iterative to string conversion from a tree element (concrete or abstract). Additionally a function ToArray will be available to convert the element to a character array and its length. Note: the resulted array is NOT zero terminated, but length terminated, because the PM can parse and the zero Unicode code point.
- ToString Recursive - a recursive to string conversion for a tree element (concrete or abstract). Additionally a function ToArray will be available to convert the element to a character array and its length. Note: the resulted array is NOT zero terminated, but length terminated, because the PM can parse and the zero Unicode code point.
- 'Expected' on error - when an error is received at runtime of the PM on a given input, if this option is turned 'on' and the error found is because of a not recognized token, then a summary list of all possible tokens and rules expected in this location is available.
- 'Expected' search mode - the algorithm used to collect the possible to recognize tokens at given location. Available is 'current stack iteration' mode that will scan the runtime stack at the time of the error and return a summary of the expected tokens. Its complexity is O(n) where 'n' is the stack depth.
- 'Found' on error - if turned 'on' when an syntax error is received it contains information about the current token recognized by the lexer that is not valid in the current parsing location.

◇ Machine - global machine options.

- Threading - single or multi threaded code to be generated. Currently the threading is available only for Microsoft Windows.
- Chained Execution (Parser, Lexer, Builder) - each module executes iterativly, regardless of the single or multihread option. Each iteration is executing as little as possible operations. This makes the loop of the iterations costly. To mitagate the effect, more operations may be executed per iteration step. Each chain execution value represent how much operation steps per iteration to execute each module (at maximum) per iteration step.
- Lexer In Parser - for grammars that have 1 token of look ahead i.e. LL(1), and this option enabled, the generated parser will have a single SAP module - the combination of the lexer and the parser modules. This makes

Figure 4.10: Tunnel Grammar StudioProject Options - Machine Tab



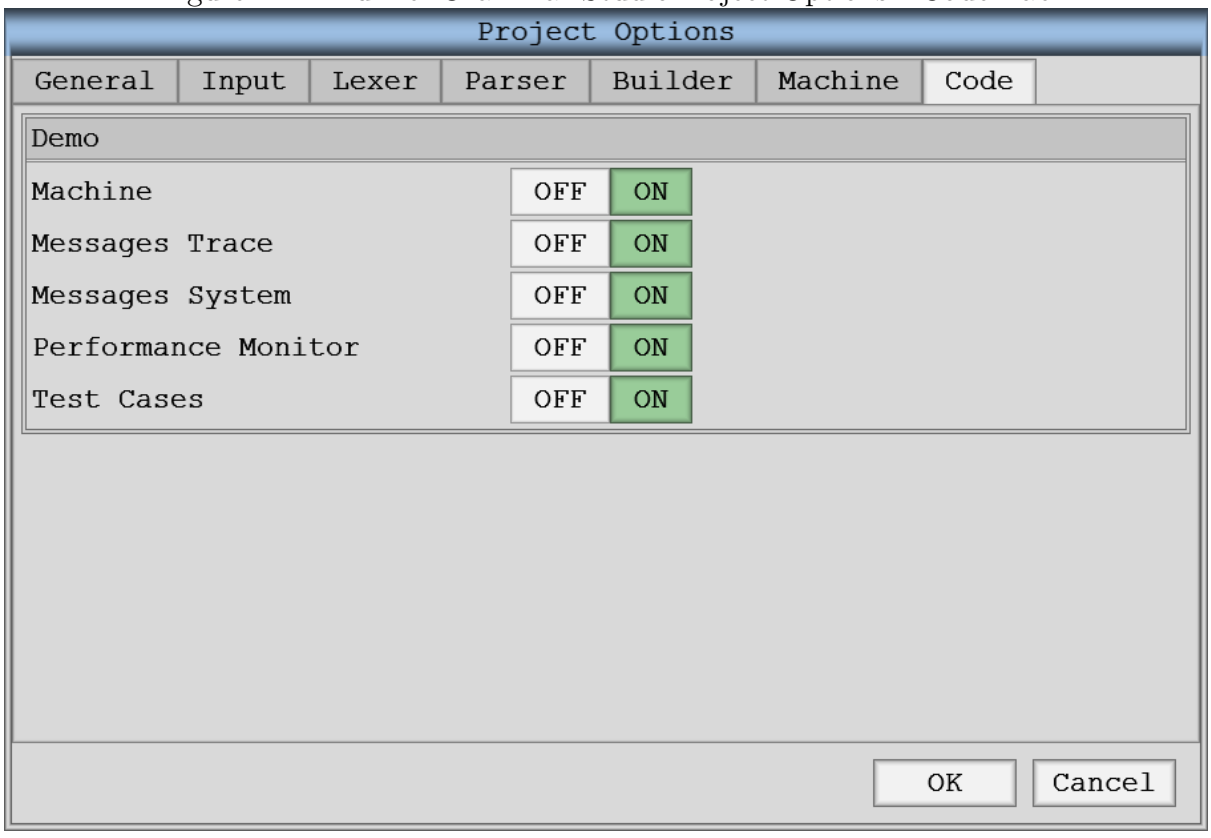
the transport of tokens to the parser more efficient and faster parsing is expected. As a consequence in a multi threaded parsers one less thread will be created.

* Builder In Parser - this makes the SAP modules bulder and parser into one SAP module. If this option is enabled, the optimizer module is removed and the commands for the construction of the syntax tree are directly used from the architect. This option is usefull for parsers with 1 look ahead symbol i.e. LL(1), because, there will no backtracking will occure during the parsing and the optimizer is not need. The optimizer have and other functionality, to group the syntax tree construction commands and send them in groups to the builder. If multithreaded parsing is used and the syntax tree is consisting of classes with many fields (large grammars), then this option MAY be better let OFF. As a consequence of this option is 'ON' in a multi threaded parsers one less thread will be created.
* Block byte size - the size of the block that is used across the whole PM.
* Lexer to Parser blocks - the RECOMMENDED amount of blocks with tokens that will be transfered at ones from the lexer to the parser.
* Parser to Builder blocks - the RECOMMENDED amount of blocks with ST commands that will be transfered from the parser (specifically the Optimizer) to the ST Builder.

* Byte Offset - the *location* structure contains byte offset from the input stream start.
* CodePoint Offset - the *location* structure contains Unicode code point offset from the input start.
* Line/Char Offset - the *location* structure contains Line and Unicode Character offset in this line from the input start. Note: this is not a column but a character offset i.e. tab symbols are counted as one.
* Tree elements locations - should the rules and groups have *start* and *final* input *location* information. The *location* information is described in section 5.6.

◇ Code - global generated source code options.

Figure 4.11: Tunnel Grammar StudioProject Options - Code Tab



* Machine - if turned 'on' this option will make the generated parser to have a demo PM generated in a separate source file.
* Messages Trave - this option controls the messages emitted from the demo parsing machine, in each event (as the syntax tree commands for example).
* Messages System - option to control the system messages related to the PM execution in the demo PM.
* Performance Monitor - generate code that measures the runtime of the different PM functionalities (parsing, iteration, to string and destruction)

and prints the results.

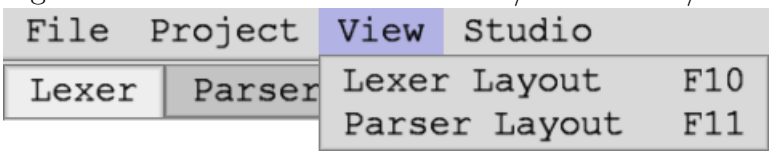
* Test Cases - generate test cases for the parser. The demo PM will automatically load the generated test cases and execute them. Positive (tests that must succsseed) and Negative (test that must fail) test cases are generated.

- Check Lexer - checking of the lexer grammar by itself for syntax or semantic errors.
- Check Parser - checking of the parser grammar by itself for syntax or semantic errors.
- Check Parser - checking of the parser and lexer grammars alone and a in combination for syntax or semantic errors.
- Compile - compile the PM based on the lexer and parser grammars with the chosen project options.

4.7.3 View

Menu for visualization of the lexer and parser grammars in figure 4.12.

Figure 4.12: Tunnel Grammar Studio/Main Menu/View

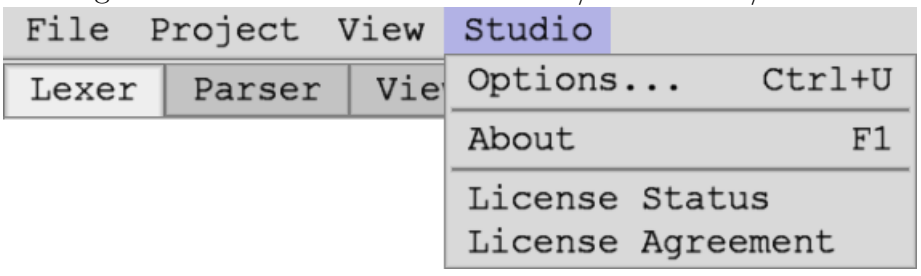


- Lexer Layout - load the lexer grammar automat layout into the view tab.
- Parser layout - load the parser grammar automat layout into the view tab.

4.7.4 Studio

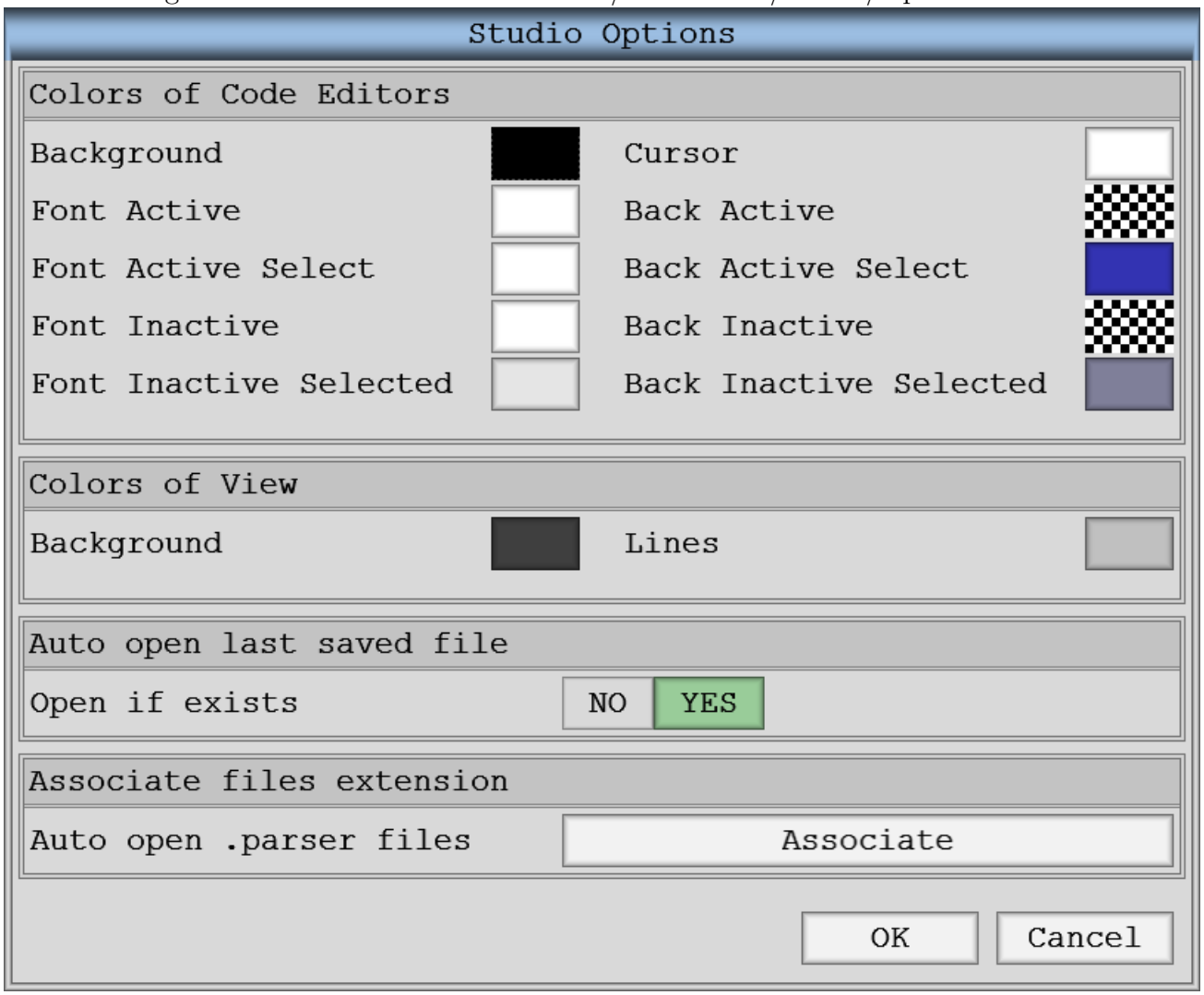
Related to the Tunnel Grammar Studio™ options and operations.

Figure 4.13: Tunnel Grammar Studio/Main Menu/Studio



- Options - related to the Studio Options as in figure 4.14.

Figure 4.14: Tunnel Grammar Studio/Main Menu/Studio/Options



- About - information about the Tunnel Grammar Studio™ currently running instance.
- License - information about the license related to the currently running Tunnel Grammar Studio™ program instance.
- Colors of Code Editors - list of colors that apply to the code editors (lexer and parser grammars as well as the debugger input) in the interface.
  - Background - the background color of the code editors. Default is black: rgba(0, 0, 0, 255).
  - Font Active - the font color when the code editor is active for input. Default is white: rgba(255, 255, 255, 255).
  - Font Active Selected - the character font color when it is selected. Default is white: rgba(255, 255, 255, 255).
  - Font Inactive - the font color of not selected characters when the editor is not active. Defaults to white: rgba(255, 255, 255, 255).

  - Font Inactive Selected - the font color of a selected character when the editor is not active for keyboard input. Defaults to near white: rgba(229, 229, 229, 255).
  - Cursor - the color of the cursor in the code editor. Defaults to white: rgba(255, 255, 255, 255).
  - Back Active - the background color of a character when it is not selected and the editor is active for a keyboard input. Defaults to a transparent color: rgba(0, 0, 0, 0).
  - Back Active Selected - the background color of a character when it is selected and when the code editor is active for a keyboard input. Defaults to white: rgba(255, 255, 255, 255).
  - Back Inactive - the background color of a character when it is not selected and the editor is not active for a keyboard input. Defaults to a transparent color: rgba(0, 0, 0, 0).
  - Back Inactive Selected - the background color of a character when it is selected and the editor is not active for a keyboard input. Defaults to a gray-blue color: rgba(127, 127, 252, 255).
- Colors of view
  - Background - the background color of the graphical components of the GUI. As for example the 'View' tab content and the Debugger visualization components. Defaults to rgba(64, 64, 64, 255).
  - Lines - the color of the lines and borders of the drawn graphical primitives around the GUI. Defaults to rgba(192, 192, 192, 255).

The users with active support may check is there newer version then the currently used in `Main Menu/About/Update Check`.

4.8 Practical limitations

There is various practical limitations that a developer is not expected or advised to reach.

Compile time:

- Rule name length - current maximum is set to 128 symbols.
- Rule names complexity - current maximum of the internal automat states is set to $2^{16} = 65536$.
- Rules per grammar - current maximum is set to 128 rules.
- Phrase value length - current maximum is set to 128 rules.

- Phrase values complexity - current maximum of the internal automat states is set to $2^{16} = 65536$.
- Nested level - current maximum is set to 32 groups.
- Repetitions - currently the maximum defined integer repetition of an element is set to $2^{32} - 1 = 4294967295$. Note that smaller values result to a less memory usage at PM runtime and that the infinity repetition is not bounded.
- NFA complexity - the first stage of the lexer grammar compilation involves a NFA construction. The maximum states limit is set to $2^{14} = 16384$.
- DFA complexity - the second stage of the lexer grammar compilation is creation of a DFA from the NFA created at the first stage. This involves expansion of all possible NFA states. The combinations may grow significantly if the grammar is too ambiguous. The maximum states limit is set to $2^{16} = 65536$.

Run time:

- Runtime maximum token length in chars - currently allowed is $2^{24} - 1 = 16777215$ Unicode characters. The actual bytes accumulated MAY be more depending from the encoding and in case of 4 bytes per encoded char, it is $2^{26} - 4$ of maximum stream byte length per token. However, its RECOMMENDED to use small values in range [32..256] set in the project options.

The values in this section are lowered for the demo version of the product. Additionally the versions MAY have other limitations.

4.9 Message Codes

100 Reference to undefined rule.

101 Phrases are not allowed in this grammar.

102 Invalid element repetition range.

200 Integer number epsilon paths - there exists more then one, but finite epsilon paths.

201 Infinite epsilon paths.

300 Follow collision for a character symbol.

301 Follow collision for a sensitive phrase with another sensitive, insensitive or general phrase from the same token type.

302 Follow collision for an insensitive phrase with another insensitive or general phrase from the same token type.

303 Follow collision for a general phrases from the same token type.

304 Follow collision for EOF system tokens.

400 Direct left recursion.

401 Indirect left recursion.

500 First/First collision for a same character symbol.

501 First/First collision for a sensitive phrase with another sensitive, insensitive or a general phrase from the same token type.

502 First/First collision for an insensitive phrase with another insensitive or general phrase from the same token type.

503 First/First collision for a general phrases from the same token type.

504 First/First collision for an EOF system token.

505 Dangling element right.

506 Direct right recursion.

507 Indirect right recursion.

600 The phrase values in the grammar are too complex.

700 The rule names in the grammar are too complex.

800 Syntax error in the grammar.

900 Too much rules.

901 Rule name too long.

902 Zero repetition not allowed.

903 Not allowed integer minimum repetition.

904 Not allowed integer maximum repetition.

905 In the repetition defined the minimum is bigger then the maximum.

906 Group nested level too deep.

907 Optional groups don't support explicit prefix repetition definition.

908 Phrase name too long.

909 Phrase value is too long.

910 Phrase values does not support ranges.

911 Value too big.

912 EOF systen token don't support explicit prefix repetition definition.

913 Range value start is bigger or equal to the final value.

1000 Unable to prepare the ABNF parser.

1001 ABNF Parser runtime error.

1002 Invalid result ABNF parser state.

1003 Missing ABNF syntax tree.

1100 Invalid NDFA minimum repetition.

1101 Invalid NDFA maximum repetition.

1102 Phrases are not allowed in this grammar.

1103 Recursion is not allowed in this grammar.

1104 The NDFA automat is too big.

1200 Too many DFA to NDFA simulator states.

1201 Rule overwrite. More then one rule can recognize an input.

Chapter 5

Target C++

For 2 valid grammars, one for the SAP Lexer and one for SAP Parser elements, a self sustaining PM is generated. The target directory of the files and the options can be changed before compilation in `MainMenu/Parser/Options`. The compilation is execute with F9 keyboard button or from `MainMenu/Parser/Compile`.

5.1 Stream Reader

The compiled machines that have source decoders are having a class with a name **stream_reader**. This class have one pure virtual function with a name **Read** that MUST be overwritten by the class extension. The full declaration of the function is `virtual READ_STATUS Read(uint8 *data, uint data_length, uint *read_length)=0;` where READ_STATUS is one of:

- `RS_OK` - indicates that **at least one** byte has been written into the data, and the total read length must be returned in argument 'read_length'.
- `RS_BUSY` - indicates that the stream is busy and the machine must stop, till more data become available. When that data is available, the PM must be prepared and run again.
- `RS_END` - the stream end was reached and no data is/will be available.
- `RS_FAIL` - there is a stream error, and the PM must stop.

5.2 Read Buffer

The compiled machines that have source decoders are having a class with a name **source_buffer**. This class receives a pointer of **stream_reader** at its compile time, that uses in its lifetime (t.e. the stream_reader MUST not be destructed before the source_buffer is). All decoders are using this buffer for fast decoding of the stream.

5.3 Source Decoders

Every PM may contain a set of source decoders, chosen in the project options.

5.3.1 ASCII

If compiled in PM, a class with a name **source_ascii** is available. Accepts bytes that directly map to an ASCII char. All other byte are substituted with the error Unicode char 0xFFFD.

5.3.2 ISO-8859-1

If compiled in PM, a class with a name **source_iso88591** is available. Directly maps every byte to an Unicode character. Other popular name of this decoder is **Latin1**.

5.3.3 Win 1252

If compiled in PM, a class with a name **source_win1252** is available. Accepts bytes that are valid in Win 1252 code page. All other byte are substituted with the error Unicode char 0xFFFD.

5.3.4 UTF-8

If compiled in PM, a class with a name **source_utf8** is available. Accepts Unicode UTF-8 encoded stream. All invalid byte sequences are replaced with the error Unicode char 0xFFFD.

5.3.5 UTF-16LE

If compiled in PM, a class with a name **source_utf16le** is available. Accepts Unicode UTF-16LE encoded stream. All invalid byte sequences are replaced with the error Unicode char 0xFFFD.

5.3.6 UTF-16BE

If compiled in PM, a class with a name **source_utf16be** is available. Accepts Unicode UTF-16BE encoded stream. All invalid byte sequences are replaced with the error Unicode char 0xFFFD.

5.3.7 UTF-32LE

If compiled in PM, a class with a name **source_utf32le** is available. Accepts Unicode UTF-32LE encoded stream. All invalid byte sequences are replaced with the error Unicode char 0xFFFD.

5.3.8 UTF-32BE

If compiled in PM, a class with a name **source_utf32be** is available. Accepts Unicode UTF-32BE encoded stream. All invalid byte sequences are replaced with the error Unicode char 0xFFFD.

5.3.9 Universal

If compiled in PM, the class named **source_universal** will be available. It accepts a default encoding argument in its constructor. It scans the first bytes of the stream and automatically determines the encoding. The check is only for the currently compiled Unicode source decoders. If a match is found, the further processing is redirected to the found decoder. If no match is found the default encoding supplied at class construction time is used.

5.4 Location

Every compiled PM have a class with a name **location**. It contains variables (and functions for their retrieval) holding information about a location in the input stream.

- `uint32 GetByte()` - the input stream byte offset where the error occurred or uint32(-1) if the byte offset is not compiled into the PM.
- `uint32 GetCodePoint()` - returns the Unicode code point offset where the error occurred or uint32(-1) if the Unicode code point is not compiled into the PM.
- `uint32 GetLine()` - returns the text line zero based index where the error occurred or uint32(-1) if the text line/char is not compiled into the PM.
- `uint32 GetChar()` - returns the text current line char (not column) zero based index where the error occurred or uint32(-1) if the text line/char is not compiled into the PM.

5.5 Events

Every compiled machine have a class with a name **events**. This class MUST be supplied in the machine constructor, and MUST NOT be deleted till the machine de-

struction. The current functions that MAY be overwritten in user defined extensions of this class are:

- `void OnDone(void)` - function called when the input was successfully recognised and there is a syntax tree constructed.
- `bool OnSyntaxError(const syntax_error &)` - this function is called from the Builder element (that may be in its own thread in case of MT PM) and contains information about an error that occurred. The function must return *true* if the machine must stop its execution after this error, and *false* if the machine will continue to search more for a valid ST.
- `void OnExplored(void)` - this function is called, when all possible combinations (related to the maximum look ahead tokens) were tested and no recognition was found.

One may create an extension of this class for file logging and another extension for GUI presentation of the client program, and switch them depending from the needs.

5.6 Syntax Error

Every instance of PM have a class with a name **syntax_error**. This function have a location information and optionally, if compiled, list of the expected items (rules or tokens) at the error location. Internals:

- `const location &Location()` - returns the location object holding information where is the error in the stream.
- `struct item` - holds a pointer for one expected item, and a pointer to the next item structure, forming a list. The last item in the list have *next* set to NULL. Available only if compiled in the PM.
- `const item *First()` - returns the first *item* structure, available only if compiled in the PM.

There are several types that MAY be available in the expected list. To determine which of this types is received one MUST check the return value of the **Type** function of the class instance.

- `expect_rule` - having a Type return value of T_RULE. Contains a public variable `const char *Name` that is the expected rule name.
- `expect_char` - having a Type return value of T_CHAR. Contains a public variable `uchar Value` that is the expected Unicode char.
- `expect_range` - having a Type return value of T_RANGE. Contains a public variables `uchar From, To` that is the expected Unicode char range **inclusive**.
- `expect_string` - having a Type return value of T_STRING. Contains a public variable `const uchar *Value` that is the Unicode character array expected, **uint Length** that is the expected character array length in chars (the array is **not** zero

terminated) and a variable **bool CaseSensitive** specifying is this string expected in a case sensitive manner.

- `expect_phrase_any` - having a Type return value of T_PHRASE_ANY. Contains a public variable `const char *Name` that is the expected lexer phrase name
- `expect_phrase_value` - having a Type return value of T_PHRASE_ANY. Contains a public variable `const char *Name` that is the expected lexer phrase name, **const uchar *Value** - an array of the expected phrase content, **uint Length** - the length in chars of the 'Value' variable (the array is **not** zero terminated) and a variable **bool CaseSensitive** specifying is this phrase value expected in a case sensitive manner.
- `expect_eof` - having a Type return value of T_EOF. Indicates that end of the stream was expected.

5.7 Parsing Machine

The PM is represented by an object with a class name **machine**. The functions defined in it are:

- `uint Prepare(void)` - called at the beginning of the stream processing. That may be at the first processing start or if the stream bytes were not available fully and the parsing was interrupted because of it, to signal continuation of the processing because new bytes become available. Returns E_OK on success and other value on fail.
- `uint RunMT(uint32 TimeOut)` - monitor the PM execution for a specific amount of time, or if uint32(-1) is used for the TimeOut argument, then the monitoring is executed till PM completion, then the control of the caller is restored i.e. this is a blocking function call. Returns E_OK if the machine did **not** complete in the given time interval, E_DONE if the machine **completed** its execution and other value for an error.
- `uint RunST(uint32 Iterations)` - execute some amount of steps of the parsing. If uint32(-1) is supplied, the machine executes till completion. Returns E_OK value if the machine did **not** complete in the requested steps, E_DONE status code if the machine **completed** its execution and other value for an error.
- `uint Interrupt(void)` - signal a stop to the multi threaded machine execution. It can be called from any thread, and will result with an error return value for the thread that currently executes RunMT function. Returns E_OK on success, or other value for an error.
- `MACHINE_STATUS GetState(state_vector &)` - this function returns the current status of the PM with additional state vector argument for internal state values. The return value is one of:
  - MS_WORKING - the machine is still working.

  - MS_NEED_INPUT - the machine does not work, because more input is need.
  - MS_COMPLETE - the machine has completed successfully.
  - MS_ERROR - an error has occurred during the execution, and the respective functions were called (as for example 'OnSyntaxError').
  - MS_FATAL: a fatal error occurred during the execution. The *state_vector* argument contains error codes for the different SAP elements. That is a system fatal error that may be generated for example in MT PM where an exception was thrown because no memory was available.
  - MS_INCOSISTENCY: the internal structures of the PM are having inconsistent values. This MUST NOT occur in practice, and MAY be a result from a memory corruption in the client program that changed the PM state.

- `basic *GetResult(void)` - returns the current syntax tree abstract root element, or NULL if no tree is available.
- `bool GetResult(%rule_class_name% **)` - one function per parser grammar rule, that returns into its single argument a pointer to the syntax tree root object, if the class types match, returns *true* as a result. If no ST root element is available *false* is returned. The object remains referenced by the PM and will be deleted by its destructor.

5.8 Files

Each generated file is named based on the PM options specified (used later as a reference %FileNamePrefix%). In the target directory, the following structure will be generated on compile time:

- %FileNamePrefix%system_bridge.h - this file, placed in the root directory of the generation, is connecting the parser code, to the client language types definitions and included headers. A file with a name %FileNamePrefix%system_bridge.h is generated at every compilation if does not yet exists. The client MAY edit the file with its own definitions and include custom headers.
- **kernel** - folder that contains general files needed for the execution of the PM, but not directly grammars related. Items inside:
  - %FileNamePrefix%system_kernel.h - this file contains the **parser_object** class.
  - %FileNamePrefix%system_kernel.cpp
- **parser** - folder that contains the SAP elements source code. Items inside:
  - %FileNamePrefix%system_parser.h
  - %FileNamePrefix%system_parser.cpp
- **tree** - the syntax tree object oriented source code one 'cpp' and one 'h' file per parser rule. Items inside (where %rule_name% defines each rule name in turn):

- %FileNamePrefix%tree_%rule_name%.h
- %FileNamePrefix%tree_%rule_name%.cpp

Each rule have its own file, for the reason the developer to include only the relevant files into the glue code. Then the compilation of the full client project is expected to be faster in long term.

For bigger grammars some compilers may not be able to compile the files generated, because they MAY be too big. The user MAY expect different files organization in the future versions, that will split the compiled code into more files.

5.9 Classes

All the code is generated inside the chosen name space, and each class name is prefixed by the chosen class name prefix (used later as %ClassNamePrefix%) both from the parser options. Because C++ class names are not allowed to contain a dash symbol, but ABNF grammars have it in the rule name, but don't have underscore and are case insensitive, all class names are generated with lower case and every dash is substituted with an underscore. As for example, the parser grammar rule with a name 'Document-Root' will generate a class name %ClassNamePrefix%document_root.

5.10 Threading

Currently the threading and synchronization is implemented for Microsoft Windows Operation System, by including "windows.h" header file and calling directly the API. For single threaded files, the header file is not included and as a consequence the generated code is expected to be not operation system dependent.

5.11 ToString

The Tunnel Grammar Studio™ have the option to collect all sub tokens symbols, of the ST generated objects, into a list. This makes easy the development of the glue code, because the developer MAY NOT specifically write code for rules or groups but extract their symbols in a Unicode char array.

5.12 Errors

The parsing errors of running PM are reported by calling the PM *events* class functions OnSyntaxError and OnExplored documented earlier in the document. It is RECOMMENDED to extend the PM *events* class and overwrite the functions for receiving the error reporting.

5.13 Memory

The SAP elements operate with blocks of raw memory allocated and freed by a shared memory manager (MM) object. This gives great runtime performance, because the data can be freely moved between the SAP elements and the very efficient memory pooling technique can be used. In case of a multi threaded PM the MM object access is synchronized and thread safe.

5.14 Using the parsing machine

In the following section many different examples and use cases are described. For full source code of this example please visit the company web page. The examples in this section are having no lexer grammar, but only a parser grammar as in 5.1.

Example 5.1: Integer list parser grammar

```
list    = integer *("," integer)
integer = 1* '0'-'9'
```

5.14.1 Beginner

The "just give me the syntax tree" is the following example. Full source code and a demo can be found at https://www.experasoft.com/en/products/grammarstudio/examples/. The example parses any grammar that have 'list' root rule. The compile options are: namespace 'demo', files prefix 'demo_'. This example demonstrates parsing machine prepare, run, check result and a syntax tree retrieval.

```
1  // include system headers
2  #include <iostream>
3  #include <conio>
4
5  // include the tree root
6  #include "demo_system_parser.h"
7
8  // define explicitly the namespace
9  namespace demo01
10 {
11
12 // prepare, run and check the parsing machine completion
13 // status; returns 'true' on success and 'false' of an
14 // error occured in which case an error message is printed
15 static bool Execute(machine &m)
16 {
17   // prepare the machine for running
18   {
19     _uint status = m.Prepare();
20     switch(status)
21     {
```

```
22        case E_OK: break; // the preparation was successfull
23
24        default:
25        {
26          // failed to prepare the machine, print the error code
27          std::printf("ERROR: Preparation error 0x%08X.\n", status);
28          return false;
29        }
30      }
31    }
32
33    // run the machine till completion (success or an error)
34    {
35      _uint status = m.RunST(-1);
36      switch(status)
37      {
38        case E_DONE: break; // the running was successfully
39
40        default:
41        {
42          // there is a runtime error
43          std::printf("ERROR: Runtime error 0x%08X.\n", status);
44          return false;
45        }
46      }
47    }
48
49    // check the status of the correct machine run
50    {
51      state_vector sv;
52      MACHINE_STATUS status = m.GetState(sv);
53      switch(status)
54      {
55        // a valid syntax tree was constructed
56        case MS_COMPLETE:
57          break;
58
59        // different no full syntax tree constructed cases
60        case MS_WORKING :
61          std::printf("ERROR: Invalid working state.\n");
62          return false;
63        case MS_NEED_INPUT:
64          std::printf("ERROR: Completed with a request for more input.\n");
65          return false;
66        case MS_ERROR:
67          std::printf("ERROR: Completed with a syntax error.\n");
68          return false;
69        case MS_FATAL:
70          std::printf("ERROR: Completed with a fatal error.\n");
71          return false;
72        case MS_INCONSISTENCY:
73          std::printf("ERROR: Completed with an inconsistency state.\n");
74          return false;
75        default:
```

```
76          std::printf("ERROR: Completed with an unknown state 0x%08X.\n",
    ↪ status);
77          return false;
78        }
79    }
80
81    // having a syntax tree
82    return true;
83  }
84
85  // parsing of a byte array
86  static void ParseBytes(const _uint8 *input, _uint length)
87  {
88    // create a memory reader stream object
89    stream_reader_memory Reader(input, length);
90
91    // create a read buffer object for this stream
92    source_buffer Buffer(&Reader, 4096);
93
94    // use the buffer as ASCII
95    source_ascii Decoder(&Buffer);
96
97    // create the class that will receive error/success
98    // message when the parsing machine runs
99    events Events;
100
101   // create a memory pool object used by the parsing machine
102   memory_pool Pool;
103
104   // create the parsing machine using the memory pool,
105   // the stream decoder, the events object and the name
106   // of the parsing grammar rule
107   machine Machine(&Pool, &Decoder, &Events, "list");
108
109   // execute the machine till an error ocures or a
110   // valid syntax tree is constructed
111   if (!Execute(Machine))
112     return; // no syntax tree was constructed
113
114   // get the result syntax tree
115   demo01_list *result;
116   if (!Machine.GetResult(&result))
117   {
118     std::printf("ERROR: Unable to get the syntax tree.\n");
119     return;
120   }
121
122   // use the tree here, currently just print
123   // a message for the success
124   std::printf("SUCCESS: The syntax tree is available.\n");
125 }
126
127 // parse a zero terminated character array
128 static void Parse(const char *input)
129 {
```

```
130   // print a header for the current parsing operation
131   std::printf("================================");
132   std::printf("\nInput:\n%s\n", input);
133   std::printf("--------------------------------");
134   std::printf("\nParse Log:\n");
135
136   // do the parsing as raw bytes, using the
137   // zero terminated character array
138   try
139   {
140     ParseBytes(reinterpret_cast<const _uint8 *>(input), strlen(input));
141   }
142   catch(const std::exception &e)
143   {
144     // in case of an exception, print it to the console
145     std::printf("*******************************");
146     std::printf("Exception: %s\n", e.what());
147   }
148
149   // operation has completed
150   std::printf("\n");
151 }
152
153 } // namespace demo01
154
155 static void PressAnyKey(void)
156 {
157   // wait key message
158   std::printf("\nPress any key to continue...");
159
160   // wait for a single button to be pressed
161   _getch();
162 }
163
164 // program entry point
165 int main(int, char *[])
166 {
167   // check was the parsing machine code compiled
168   // with the expected integer type sizes
169   try { demo::check_compiled_options(); }
170   catch(const std::exception &e)
171   {
172     std::printf("Exception: %s\n", e.what());
173     return -1;
174   }
175
176   // do parsing of a correct input
177   demo01::Parse("54,12,74561");
178
179   // do parsing of a wrong input
180   // (because char 'x' is not a number)
181   demo01::Parse("1,9,x,4");
182
183   // wait for a key
184   PressAnyKey();
```

```
185   return 0;
186 }
```

5.14.2 Advanced

To report properly syntax errors found in the input functions in the **events** class must be overwritten by extending the class. Full source code and a demo can be found at https://www.experasoft.com/en/products/grammarstudio/examples/. This example adds **CustomEvents** class in the *Basic* example and uses it instead of **events** to receive and print the runtime parsing events:

```
1  // code inside the namespace
2
3  // direct printing of a UTF32 character array
4  static void sprintf_utf32(const _uchar *text)
5  {
6    while(*text) std::printf("%c", *text++);
7  }
8
9  //-------------------------------------------------------------
10 //       CustomEvents extends events
11 //
12 // this class receives events during
13 // the parsing machine runtime
14 //
15 //-------------------------------------------------------------
16 class CustomEvents : public events
17 {
18   public:
19     virtual      void     OnSyntaxError(const syntax_error &);
20     virtual      void     OnExplored(void);
21     virtual      void     OnSuccess(void);
22 };
23
24
25 // event function called for each syntax error found
26 void CustomEvents::OnSyntaxError(const syntax_error &e)
27 {
28   // error header
29   std::printf("ERROR: ");
30
31   // conert the syntax error do a character array
32   _uchar *text = e.ToArray(NULL);
33
34   // print
35   sprintf_utf32(text);
36
37   // free the memory
38   delete[] text;
39
40   // line termination
41   std::printf("\n");
42 }
```

```
43
44 // event function called when exploration has completed
45 // and there is no syntax tree constructed
46 void CustomEvents::OnExplored(void)
47 {
48   std::printf("INFO: Explored.\n");
49 }
50
51 // event function called a syntax tree is constructed
52 void CustomEvents::OnSuccess(void)
53 {
54   std::printf("INFO: Success.\n");
55 }
56
57 // use the new events, by replacing the default system 'events'
58 static void ParseBytes(const _uint8 *input, _uint length)
59 {
60   .......
61
62   CustomEvents Events;
63
64   .......
65 }
```

5.15 Expert

After a syntax tree element is available, it can be iterated as much as need before its released. The following glue code can be added to the advanced example to view the resulted syntax tree. Full source code and a demo can be found at https://www.experasoft.com/en/products/grammarstudio/examples/.

```
1 // add the exceptions header
2 #include <exception>
3
4 // add the 'integer' rule header
5 #include "demo_tree_integer.h"
6
7 .....
8
9 static void InvalidTree(void)
10 {
11   throw std::runtime_error("Bad formed syntax tree");
12 }
13
14 static void PrintTree_Integer(demo_integer *i, _uint depth)
15 {
16   // ident based on the depth
17   for(_uint k=0; k<depth; k++)
18     std::printf(" ");
19
20   // check the integer rule concatenation index
21   switch(i->Index())
```

```
22   {
23     case 0:
24     {
25       // iterate the digits found and print them
26       for(demo::list<_uint>::Item *e = i->alt.c0->h0.First;
27           e; e = e->Next)
28         std::printf("%c", e->Data);
29     } break;
30
31     // the integer rule have only one concatenation
32     default: InvalidTree();
33   }
34
35   // end
36   std::printf("\n");
37 }
38
39 static void PrintTree_String(demo::string &s, _uint depth)
40 {
41   // ident based on the depth
42   for(_uint k=0; k<depth; k++)
43     std::printf(" ");
44
45   // print all chars in the string
46   sprintf_utf32(s.Text(), s.Length());
47
48   // end
49   std::printf("\n");
50 }
51
52 static void PrintTree(demo_list *list)
53 {
54   // header
55   std::printf("TREE:\n");
56
57   // check the concatenation index of the 'list' rule
58   switch(list->Index())
59   {
60     case 0:
61     {
62       // print the first rule integer 'r0integer'
63       // that is the start of the 'list' rule
64       PrintTree_Integer(list->alt.c0->r0integer, 0);
65
66       // iterate all second element that is a list of groups
67       typedef demo_list::ALT::C0 *list_C0_t;
68       for(demo::list<list_C0_t>::Item *e = list->alt.c0->g1.First;
69           e; e = e->Next)
70       {
71         // entering in depth
72         std::printf("(\n");
73
74         // check the concatenation index
75         switch(e->Data->Index)
76         {
```

```
77            case 0:
78            {
79              // get a pointer to the group's first concatenation
80              list_C0_t::A1::C0 *groupC0 = e->Data->Value.c0;
81
82              // print the first item string 's0'
83              PrintTree_String(groupC0->s0, 1);
84
85              // print the second item rule integer 'r1integer'
86              PrintTree_Integer(groupC0->r1integer, 1);
87            } break;
88
89            // the first group does have one concatenation only
90            default:
91              InvalidTree();
92          }
93
94          // leaving the depth
95          std::printf(")\n");
96        }
97      } break;
98
99      // the grammar have only a single concatenation inside
100     // any other valid is invalid
101     default: InvalidTree();
102   }
103 }
104
105 static void ParseBytes(const _uint8 *input, _uint length)
106 {
107   ......
108
109   demo_list *result;
110
111   ......
112
113   // print the tree when available
114   PrintTree(result);
115 }
```

Chapter 6

Use Cases

In this chapter are defined grammars forming valid PM, starting from more simple to more complex cases.

6.1 Text Splitter

To recognize text expressions from the input one may easy write the lexer and parser grammars in examples 6.1 and 6.2. The lexer grammar reads as: a 'word' is one or more case insensitive 'a'-'z' letters and a number is one or more digits. The parser grammar reads as: a text is one or more expressions that each is one or more words that in between have an OPTIONAL comma and a REQUIRED space' and MUST finish with a full stop. A valid input for a PM generated from this two grammars will be: `"Use case 1: Hello World"`. Additionally one may use only a parser grammar to achieve the same as in example 6.3.

Example 6.1: Text splitter lexer grammar

```
1 word   = 1* ('A'-'Z' / 'a'-'z')
2 number = 1* '0'-'9'
```

Example 6.2: Text splitter parser grammar

```
1 text       = 1*expression
2 expression = ({word} / {number})
3              *(0*1 "," " " ({word} / {number}))
4              "."
```

Example 6.3: Text splitter standalone parser grammar

```
1 text       = 1*expression
2 expression = (word / number)
3              *(0*1 "," " " (word / number))
4              "."
5 word       = 1* ('A'-'Z' / 'a'-'z')
6 number     = 1* '0'-'9'
```

However the two approaches will have different syntax trees. The first way with the 2 grammars, is expected to run faster in the general case, because the Lexer operates faster then the Parser, because no context information is recorded, as a consequence each token 'word' and 'number' will be represented by a single string type of object. In the second approach with single grammar, the context information is recorded, that will take more memory and time to be recognized and constructed. As a general rule, everything that represents single tokens SHOULD be moved into the Lexer grammar.

6.2 Mathematical Expression

In many programming languages mathematical expressions are build in to match the most popular infix notation. Such recognition can be made as the examples 6.4 and 6.5.

Example 6.4: Math expression lexer grammar

```
1 varname    = 1* ('A'-'Z' / 'a'-'z')
2 integer    = 1* '0'-'9'
3 float      = 1* '0'-'9' "." * '0'-'9'
```

Example 6.5: Math expression parser grammar

```
1 expression     = addition
2 addition       = multiplication
3                  *(("+" / "-") multiplication)
4 multiplication = value *(("*" / "/") value)
5 value          = {integer} /
6                  {float} /
7                  {varname} /
8                  "(" addition ")"
```

As defined these grammars will generate PM which will be LL(1) and accept the following expressions:

- 2*x
- 5*(alpha+60*2)
- 2*(21-(55/y)*2)
- and so on...

If one wants to introduce space in between the digits and signs, one may change the parser grammar as in this example 6.6.

Example 6.6: Math expression parser grammar with white space

```
1  expression      = *WSP addition
2  addition        = multiplication
3                    *(("+" / "-") *WSP multiplication)
4  multiplication  = number *(("*" / "/") *WSP number)
5  number          = ({integer} /
6                     {float} /
7                     {varname} /
8                     "(" *WSP addition ")")
9                    *WSP
10 WSP             = %x20 / %x9
```

The new introduced WSP rule is short for 'white space' (a space or a horizontal tab) and will be OPTIONAL at the beginning of the math expression and after each symbol and a phrase token - all currently in the 'number' rule. The priority of the multiplication is higher and comes naturally from the rules relation. These grammars form a LL(1) parser that can parse linearly input as:`24 / 4.4 + 11 * (7.8 - 2.)`.

6.3 Log File Reader

Many computer programs are saving some kind of log files for the started or completed operations, for the errors and warnings occurred during the program runtime. This logs are often specifically formated for the company needs. One may define grammars (as example 6.7 and 6.8) that read this logs without the need to create a specific parser by hand, if one have a need to make a statistics on the log or another automatic analysis. Additionally when the log structure changes, one have to just update the grammars defined and recompile the PM. That is expected to save a lot of time from hand coding the parser at every change.

Example 6.7: Log reader lexer grammar

```
1 string = '"'
2           *(%x20-%x21 /
3             %x23-%x5B /
4             %x5D-10FFFF /
5              '\' ('\' / '"'))
6           '"'
7 word   = ('A'-'Z' / 'a'-'z' )
8           *('A'-'Z' / 'a'-'z' / '0'-'9' / '-' / '_')
9 number = 1* '0'-'9' ["." * '0'-'9']
```

Example 6.8: Log reader parser grammar

```
log  = *line
line = item *("," item) CRLF
item = {string} / {word} / {number}
CRLF = %xD %xA
```

In words, the example lexer grammar is recognizing simple tokens as:

- **string**: starting and ending with double quotes with a possible escaping '\' char followed by '\' or double quote
- **number**: an integer or a float
- **words**: starting with a letter and optionally continuing with a letter, digit, dash or underscore.

The parser grammar is recognizing an input of lines terminated with the Internet standard line terminator. Each line MUST have one or more items separated by a comma, and each item is one of the lexer defined tokens: string, word or a number. No white space is permitted inside the log file, but it can be added in the same way as in the math expression use case in a section 6.2

6.4 Simple Markup Language

Many companies have option files that have to be human and machine readable. To define simple markup language for this purpose one may use as a base the Extended Markup Language (XML)[4] as in examples 6.9 and 6.10. A valid input to a PM generated by these grammars would be example 6.11.

Example 6.9: Simple ML lexer grammar

```
comment-start = "<!--"
comment-final = "-->"
tag           = "<" 0*1 "/"
word          = 1* ('A'-'Z' / 'a'-'z')
```

Example 6.10: Simple ML parser grammar

```
document  = *WSP tag *WSP
tag       = tag-start content tag-final
tag-start = {tag, "<"}  *WSP {word} *WSP '>'
tag-final = {tag, "</"} *WSP {word} *WSP '>'
content   = *(tag / %x21-3B / %x3D-10FFFF / {word} / WSP)
WSP       = %x9        / ; tab
            %x13 %x10  / ; new line
            %x20       / ; space
            {comment-start}
            *(%x0-10FFFF / {word} / {comment-start} / {tag})
            {comment-final}
```

Example 6.11: Simple ML input

```
<options>
<-- this is an option file with one string -->
<string>hello word!</string>
</options>
```

The defined examples, construct a language that accepts white space before and after any word and symbol. Additionally the comment text is syntactically available that is defined to "everything between `<--` and `-->` sequences".

6.5 Simple C/C++

If one wants to create a script language, one may base it on the C/C++ programming languages. On the first phase of the parsing the syntax is checked with the PM, on a second phase the semantical meaning of the parsed code could be checked and finally an interpreter or a generator to another language (possibly Assembler) could execute the syntactically and semantically valid parsed input. Examples 6.12 and 6.13 are containing grammars that can construct such a C similar PM.

Example 6.12: Simple C/C++ lexer

```
comment = "/*" / "*/" / "//"
CRLF    = %x13 %x10
keyword = %s"const" / %s"if"
```

Example 6.13: Simple C/C++ parser

```
1  document        = *WSP *function
2  function        = %s"function" 1*WSP name *WSP
3                    func-args scope
4  func-args       = '(' *WSP [argument-list] ')'
5  func-call       = '(' *WSP [operation-list] ')'
6  argument-list   = argument 1*WSP *(',' 1*WSP argument 1*WSP)
7  argument        = [{keyword, "const"} 1*WSP] type-and-name
8  type-and-name   = name 1*WSP name
9  operation-list  = operation *(',' 1*WSP operation)
10 scope           = '{' WSP *declaration "}" 1*WSP
11 declaration     = decl-var / decl-if
12 decl-var        = type-and-name *WSP ['=' *WSP operation] ';' *WSP
13 decl-if         = {keyword, "if"} *WSP '(' *WSP operation ')' *WSP
14                   scope [{keyword, "else"} *WSP scope]
15 operation       = addition
16 addition        = multiplication
17                   *(("+" / "-") *WSP multiplication)
18 multiplication  = value *WSP
19                   *(("*" / "/") *WSP value *WSP)
20 value           = name 0*1 func-call /
21                   number /
22                   "(" *WSP operation ")"
23 name            = ('a'-'z' / 'A'-'Z')
24                   *('A'-'Z' / 'a'-'z' / '0'-'9' / '_')
25 number          = 1* '0'-'9' ['.' * '0'-'9']
26 WSP             = %x9 / %x20 / {CRLF} /
27                   line-comment / text-comment
28 line-comment    = {comment, "//"} *(%x0-10FFFF / {comment}) {CRLF}
29 text-comment    = {comment, "/*"}
30                   *(
31                     %x0-10FFFF /
32                     {CRLF} /
33                     {comment, "/*"} /
34                     {comment, "//"}
35                    )
36                   {comment, "*/"}
```

These grammars form a LL(1) grammar for parsing C/C++ like source code.

The semantic analysis is not defined into the grammar, as for example is it an error if one calls a function that is not defined, or calls a function that is defined later in the source code. Because the parsing phase is completed fully before the semantic analysis is started, one may simply permit calling of functions that are defined later in the code, that is one of the advantages of having the parsing phrase separate of any other phase.

Bibliography

[1] Key words for use in RFCs to Indicate Requirement Levels, https://tools.ietf.org/html/rfc2119, 2018/01/01.

[2] Augmented BNF for Syntax Specifications: ABNF, https://tools.ietf.org/html/rfc5234, 2018/01/01.

[3] CaseSensitive String Support in ABNF, https://tools.ietf.org/html/rfc7405, 2018/01/01.

[4] Extensible Markup Language (XML), https://www.w3.org/TR/xml/, 2018/01/01.