Gold Parser Engine

1. Contents

2. Intro

Here is a brief explanation of how parsers work. I strongly advise that anyone trying to understand these blueprints should have a prior knowledge of the subject, otherwise one can find the provided information frustratingly insufficient, as this is oversimplified and just scratching the surface.

A parser is a program that takes a string input and produces a syntax tree structure in form of some classes that can later be manipulated programmatically. It is used whenever some text is constructed following some rules and that text must be made sense of in order to extract data from it or be manipulated, following said rules. Syntax highlighters in integrated development environments use this in order to be able to make sense of the input source code and color different words properly. Also, programing language compilers use parsing as those tools are essentially translators from a high-level language to assembly language. Different programming language translators use parsing as well, for example, a program that converts C# or JavaScript source code to Java source code. Also, our human languages can be parsed in order to extract data. Bottom line - whenever source code manipulation must be performed a well constructed and efficient parser is needed.

A parser follows a set of rules called a language grammar to parse the given input text. This grammar might be inbuilt in the parser, provided to the parser as text or as precompiled grammar file, like in our case. Grammars are written in some form - this varies from a parser to parser. It is some metalanguage but in modern parsers, it is almost always some form of BNF or EBNF implementation. BNF stands for Backusâ€“Naur Forms and EBNF stands for Extended Backusâ€“Naur Forms.

There are different kinds of parsers, and most have some limitations on the grammar syntax allowed. The most popular parsers used today are of the LALR type, which also has some limitations but is really powerful and fast, and sufficient enough for most of the current programming languages. There are parser types that don't have grammar limitations but are significantly slower and thus rarely used. Also, various grammars for LALR parser engines are available online for free.

A parser doesn't follow the grammar directly, but instead a set of rules that dictate actions to be taken in different situations. That being said, a grammar needs to be verified to make sure it complies with the restrictions, and most of the time the grammar is automatically optimized and then converted to a set of such rules, and often written to a file. Because of that, most modern parsers are not built by programmers anymore. Instead, one writes a grammar and compiles it to a set of rules using a tool, in order to use the grammar with universal parser engine, or the grammar compiler outputs the parser source code in the desired language with the inbuilt set of rules in it, instead.

Almost always in modern parsers parsing is done in two stages - a lexical and syntactical stage, and thus a parser consists of two tools - a lexical analyzer, also called a Tokenizer that produces units called lexemes or Terminals - that is the equivalent of words in our spoken language(English for example). The second tool is the syntactical analyzer that creates syntactical units that are also called Nonterminals - that is the equivalent of sentences and probably bigger units like paragraphs and chapters of a book in our spoken language(English for example). Both tools need a different set of rules to act upon. Those are both produced when the grammar is compiled and put in a single file. Also, in some limited cases, one would only need a lexical analyzer to be created, and many tools and engines allow that as well.

So, what we have here is the schematics for the "Gold Parser Engine" - (2 stage - lexer/parser DFA, LALR(1)parser), without the grammar compiler (in another document), and it consists of the following :
1. The Token system is the parse tree. Getting the parse tree structure is the goal and it(the token structure) can be manipulated in code. It holds various information about the parsed source - a unit from this tree

contains the information of the position of its elements in the original source code. It can be used to construct a more convenient AST structure(stands for Abstract Syntax Tree) that can be manipulated easier. It can also be traversed to source code after all the desired manipulations are done.
2. The grammars - DFA for the lexer and LALR for the parser are the rules to be used, in our case precompiled in grammar files. I will be using the word grammar somehow interchangeably in this text to refer to either the set of Backusâ€“Naur Forms or the set of rules compiled from the Backusâ€“Naur Forms.
3. The engine itself consists of the Lexer and Parser tools that, using the set of grammar rules parse the input string to syntax units.
4. The deserialization system is simply used to read the compact structure of a compiled grammar file and turn it to set of rules in the form of different classes the parser and lexer understand.

3. Parse Tree (The Token System).

A Parse tree is a structure that can be accessed recursively.
In most cases, it consists of a parent element class from which terminals and nonterminals inherit. Nonterminals consist of terminals and other nonterminals, while terminals are the basic building block. Both should contain the start and end location they represent in source code string. Also, both must be easily traversable to source code - for example by just calling .ToString() on a parse tree will return the source code the tree represents.

AST trees are similar, but they are more specific and easier to manipulate - For example an AST tree might consist of different classes for different language constructs - for example, namespace object that accepts 'class' and 'struct' objects as children and probably has methods like '.AddClass(Class child)' and '.AddStruct(Struct child)', but not methods, fields, or expression objects. It can also contain methods like '.RenameChild(string oldname, string newname)'. AST trees are more prohibitive and are language specific. They are also heavier and more resource intensive to work with. For that reason, lexers and parsers work with parse trees that and programmers prefer to work with ASTs.

In the case of the Gold Parser Engine, the base source unit is the "Symbol". Also, reduction rules are part of the Nonterminal symbols, which seems a bit unusual, at least for me. It is also worth noting that no AST system is implemented in the engine

A rule is a building block of a grammar - Each grammar consists of rules, also called productions. A rule has a left-hand side and right-hand side. For example "<A> ::= <B> 'text' <C> " which means "Non terminal 'A' is Non terminal named 'B', followed by Terminal 'text', followed by Non terminal named 'C'". Those productions are compiled into rules and in this case, we have a Shift-reduce parser so those are called Reduction rules.

The symbol system represents a simple lexeme, on top of which more complex Token system is implemented. Each symbol has id and text - that is the Name field. Some don't require text as it is the same every time.

Symbol == Symbol

Symbol != Symbol

SymbolNonterminal : Symbol	
.int	***Id***
.string	***Name***
.ToString()	*'<name>'*
SymbolTerminal : Symbol	
.int	***Id***
.string	***Name***
.ToString()	*'name'*
SymbolWhiteSpace : Symbol	
.int	***Id***
.string	***Name*** *='(Whitespace)'*
.ToString()	*'(Whitespace)'*
SymbolEnd : Symbol	
.int	***Id***

.string	***Name*** *='(EOF)'*	
.ToString()	*'(EOF)'*	
SymbolCommentStart : Symbol		
.int	***Id***	
.string	***Name*** *='(Comment Start)'*	
.ToString()	*'(Comment Start)'*	
SymbolCommentEnd : Symbol		
.int	***Id***	
.string	***Name*** *='(Comment End)'*	
.ToString()	*'(Comment End)'*	
SymbolCommentLine : Symbol		
.int	***Id***	
.string	***Name*** *='(Comment Line)'*	
.ToString()	*'(Comment Line)'*	
SymbolError : Symbol		
.int	***Id***	
.string	***Name*** *='(ERROR)'*	
.ToString()	*'(ERROR)'*	

The Rule consists of the symbols that can be reduced to another symbol.

Rule		
.int	***Id***	
.SymbolNonterminal	***Lhs***	*Left hand side.*
.Symbol[]	***Rhs***	*Right hand side.*

A location object - defines a token position in source code (in the Tokenizer system).

Location	
.int	***Position***
.int	***LineNr***
.int	***ColumnNr***
.NextLine()	
.NextColumn()	

The Token system is the actual parse tree. The abstract class Token representing both terminal and nonterminal tokens is the base of the parse tree.

TerminalToken : Token		
.string	***Text***	
.SymbolTerminal	***Symbol***	
.Location	***Location***	
.ToString()	*'Text'*	
NonterminalToken : Token		
.Token[]	***Tokens***	
.Rule	***Rule***	*The rule that caused the reduction.*
.SymbolNonterminal	***Symbol***	
.ToString()	*'name = [Text1Text2Text3]'*	

---

4. The Grammar system - DFA/LALR.

A DFA based lexer will start at a state named with a symbol and will go to a different next state depending on what character is red. The DFA is a structure representing a set of rules and a state-holder.
Each State has Transitions that can be executed. Each Transition has one or more characters - each character can trigger the transition to the destination state. So, when the DFA is asked to react to a character it looks at the current state it's in, looks at all the possible transitions for this state and searches for one that will be triggered by the given character. When found, the current state is set to the one that the triggered transition points at.
Some states are accepting states. This means that the string red so far can be legally interpreted as a token. Some lexers will acknowledge that and return the token, while others, like this, for example, will wait until no other transitions are possible and only will then return the longest possible token. This behavior varies from lexer to lexer and can be set sometimes, but in this case is not settable and the longest accepted string will be returned. At this point, the DFA is reset to the start state.
On the other hand, why the field containing all the states ("States") is necessary I don't know, it might not be necessary at all.

DFA			
.State		***StartState***	
.int		***Id***	
.Transition[]		***Transitions***	*A transition (edge) between DFA states.*
.State		***Target***	*The target state.*
.char[]		***CharSet***	*The criteria for the transition.*
.State		***CurrentState***	*The current state in the DFA.*
.int		***Id***	
.Transition[]		***Transitions***	*A transition (edge) between DFA states.*
.State		***Target***	*The target state.*
.char[]		***CharSet***	*The criteria for the transition.*
.State[]		***States***	
.int		***Id***	
.Transition[]		***Transitions***	*A transition (edge) between DFA states.*
.State		***Target***	*The target state.*
.char[]		***CharSet***	*The criteria for the transition.*
.(State[], State)		***states, startState***	
.Reset()			*Sets DFA to starting state to get a new token.*
.GotoNext(char)	***-> State***		*Goto next state depending on an input character.*

The LALR set of rules is pretty much the same as the DFA - DFA can output the new state for a character or no possible state meaning an error in the input being parsed - the LALR dictionary is used to get a new state for a Symbol.
There are different actions that guide the parser what to do, and "Action" is the base class for those.

AcceptAction : Action		
.SymbolTerminal	***Symbol***	*The symbol that a token must be for it to be accepted.*
ShiftAction : Action		
.SymbolTerminal	***Symbol***	*The criteria for this action to be done.*
.LALRState	***State***	*The new current state for the LALR parser.*
GotoAction : Action		
.SymbolNonterminal	***Symbol***	*The criteria for this action to be done.*
.LALRState	***State***	*The new current state for the LALR parser.*
ReduceAction : Action		
.SymbolNonterminal	***Symbol***	*The criteria for this action to be done.*

.Rule		rule	The rule to reduce the tokens.
LALRState			
.int		Id	
.Dictionary<Symbol, Action>		Actions	

5. Lexer/Parser engine.

The Lexer is used to create Terminal tokens from the source sequentially, one at a time. While some lexers will tokenize the entire string in order to be parsed later, most act like a stream, retrieving one token at a time and advancing the input. The input can also be skipped to or after a character so, for example, in situation where comments are ignored one can skip to after a new line character.
The lexer needs it's DFA and an input string in order to function. The input string is set into string reader wrapper and parsed one token at a time following the DFA instructions.
The "DFAInput" is basically a string reader, and the "Location" is a structure that identifies the position in the source code.

StringTokenizer		The Lexer	
.GetInput()	-> State	Gets the input string for the tokenizer.	
.SetInput(string)		Sets the input string for the tokenizer.	
.GetCurrentLocation()	-> Location	Gets the location where the tokenizer is.	
.SetCurrentLocation(Location)		Sets the location where the tokenizer is.	
.(DFA)			
_DFA		_dfa	
_DFAInput		_input	
.RetrieveToken()	-> TerminalToken		
.SkipToChar(char)	-> bool		
.SkipAfterChar(char)	-> bool		
DFAInput		Input(string) reader.	
.string		Text	
.Location		Location	
.int		Position	Location.Position
.ReadChar()	-> char	Reads a character and updates location.	
.ReadCharNoUpdate()	-> char	Reads a character without updating the location.	
.SkipToChar(char)	-> bool	Skips characters in the input until a certain character is found.	
.SkipAfterChar(char)	-> bool	Skips characters in the input until after a certain character is found.	
.IsEof()	-> bool	Determines if the input has reached the end.	
Location			
.int		Position	The zero-based position.
.int		LineNr	The zero-based line number.
.int		ColumnNr	The zero-based column number.
.NextLine()			
.NextColumn()			

RetrieveToken() ->TerminalToken

Reset the DFA before new token can be retrieved. Preserve starting location.

If position > length return EOF 'TerminalToken'.
Else read a char from input, advance dfa with this char and get the resulting state.
while resulting dfa state is not null, if the dfa state is accepting - preserve the current State and input location and symbol.
if input.IsEof() set state to null else get next state. Continue, until the dfa returns null state.
If no accepting State and input location had been set we have an error -
return new ERROR 'TerminalToken' with the string taht's been red and location. get string been red by substring start location from the input.
Else set input Location to the one preserved with the last accepting State. Retrieve text by substringing.
return new 'TerminalToken' with the string been red and location and the accepted symbol.

The parser generates Nonterminal structure(the parse tree) in the form of a single Nonterminal(representing the root of the tree) and it's child nodes as branches, while the Terminal child nodes are the leaves of the tree. The parser needs a tokenizer to stream tokens from and a set of rules to follow that come in the form of the LALRState "StartState".
The parser also contains a collection of expected symbols that can be retrieved should an error occur, and a set of events that can be subscribed to, in order to monitor the parsing process - basicly when the parser performs an action a corresponding event is raised. A consumer code can subscribe to these events in order to log the process or specify some behavior - like whether the parsing should continue in case of a specific error(what is meant by error is a token stream not complying to the grammar rules, of course not software bugs or exceptions).
Below are the EventArgs structures that are used when subscribing to those events.

TokenReadHandler OnTokenRead - Token has been read and will be parsed. Accepts "TokenReadEventArgs".		
TokenReadEventArgs		
.TerminalToken	***Token***	*The terminal token that will be processed.*
.bool	***Continue***	*If the parseing should continue after this event.*
ShiftHandler OnShift - Called when a token is shifted onto the stack. Accepts "ShiftEventArgs".		
ShiftEventArgs		
.TerminalToken	***Token***	*The terminal token that is shifted onto the stack.*
.LALRState	***NewState***	*The state that the parser is in after the shift.*
ReduceHandler OnReduce - Called when tokens are reduced. Accepts "ReduceEventArgs".		
ReduceEventArgs		
.Rule	***Rule***	*The rule that was used to reduce tokens.*
.NonterminalToken	***Token***	*Consists of tokens that has been reduced by the rule.*
.LALRState	***State***	*The state after the reduction.*
.bool	***Continue***	*If the parse process should continue after this event.*
GotoHandler OnGoto - Called when a goto occurs (after a reduction). Accepts "GotoEventArgs".		
GotoEventArgs		
.SymbolNonterminal	***Symbol***	*The symbol that causes the goto event.*
.LALRState	***NewState***	*The state after the goto event.*
AcceptHandler OnAccept - Called if the parser is finished and the input has been accepted. Accepts "AcceptEventArgs".		
AcceptEventArgs		
.NonterminalToken	***Token***	*The fully reduced nonterminal token.*
TokenErrorHandler OnTokenError - Called when the tokenizer cannot recognize the input. Accepts "TokenErrorEventArgs".		
TokenErrorEventArgs		
.TerminalToken	***Token***	*The error token.*
.bool	***Continue***	*If true continue and ignore current token.(= false)*

ParseErrorHandler OnParseError - Called when the parser has a token it cannot parse. Accepts "ParseErrorEventArgs". The ContinueMode enum is used to specify behaviour. ContinueMode.Stop - not try to parse the rest of the input. ContinueMode.Insert will insert the user provided Terminal in order to patch a missing token and fix the production, continuing with the token caused the error. This is the purpose of the "NextToken" field - to provide this terminal. ContinueMode.Skip will just ignore the current bad token and proceed to parse the input as if nothing happened.

ParseErrorEventArgs

.TerminalToken	***Unexpected***	*The token that caused this parser error.*
.Symbol[]	***Expected***	*The symbols that were expected by the parser.*
.ContinueMode	***Continue***	*= ContinueMode.Stop*
.Stop		
.Insert		
.Skip		
.TerminalToken	***NextToken***	*= null*

CommentReadHandler OnCommentRead - Called when a comment section has been read. Accepts "CommentReadEventArgs".

CommentReadEventArgs

.string	***Comment***	*The comment, including comment characters.*
.string	***Content***	*The content of the comment.*
.bool	***LineComment***	*Determines if it is a line or block comment.*

LALRParser

.TokenReadHandler	***OnTokenRead***	
.ShiftHandler	***OnShift***	
.ReduceHandler	***OnReduce***	
.GotoHandler	***OnGoto***	
.AcceptHandler	***OnAccept***	
.TokenErrorHandler	***OnTokenError***	
.ParseErrorHandler	***OnParseError***	
.CommentReadHandler	***OnCommentRead***	
.bool	***TrimReductions***	
.StoreTokensMode	***StoreTokens***	*Always, NoUserObject, Never*
.(IStringTokenizer, State[], State, Symbol[])	*tokenizer, states, startState, symbols*	
_StringTokenizer	***_tokenizer***	
_State[]	***_states***	
_State	***_startState***	
_Stack<State>	***_stateStack***	
_Stack<Token>	***_tokenStack***	
_TerminalToken	***_lookahead***	
.Parse(string) ***-> NonterminalToken***	***input***	*Parse the input with tokens and rules.*

Parse(string) ->NonterminalToken

Reset everything - including setting new laxer with the input string.
While a flag indicates to continue parsing - get lookahead 'TerminalToken', and if it is not null parse it.
After parsing it If the accepted flag is set to return the 'NonterminalToken' - the object on top of the token stack (it represents the parse tree root).
- Get Lookahead: Set the "lookahead" field if we don't have a comment or whitespace token, else try to handle it and try with another token. Try to perform OnTokenRead if it had been set.
- Handle comment by: skipping the tokenizer to the first '\n' symbol and if OnCommentRead event is present call it with start and end locations of the comment.
- Handle multiline comment: start by counting comment start and comment ends until out of comment, but if EOF symbol is met - set EOF token on the token stack and fire error. perform 'OnCommentRead' if it had been set.

- Handle whitespace by doing nothing.
- Handle error - if 'OnTokenError' event had been assigned execute it and get from its 'TokenErrorEventArgs' argument wether execution should continue 'true' or not 'false', else return false - handling failed, execution should stop.
- ParseTerminal - Peek the LALR-State on from the state stack, and get the action corresponding to the lookahead symbol. If the action is shift - doShift, if Reduce - do reduce, if Accept - do Accept, else stop parsing and fire OnParseError if set.
- DoShift by pushing the state of the shift action on the state stack, push the token on the token stack, reset lookahead field for another round and execute the OnShift event if it had been set.
- DoAccept by stopping parsing, set accept flag to true and execute OnAccept event if it had been set.
- DoReduce by - Get Rule length for this ReduceAction. If 'TrimReductions' is set and the rule right side consists of only one single 'NonterminalToken' then remove this state from the state stack and peek another state to be set as current state. Else for as many times as the number of symbols in the right side of the above-mentioned rule - remove a state from the state stack and remove Tokens from the token stack and backward-fill an array with these tokens. Then create 'NonterminalToken' with this array and the 'Rule' being used and push it on the token stack. Also set the current state by Peaking it from the state stack. Also if OnReduce is set - call OnReduce and call DoReleaseTokens. Get 'GotoAction' with the currentState and the left hand side of the used rule. If there is no GotoAction with the symbol we have invalid action table and throw exception. else perform doGoto.
- DoGoto by pushing the state of the 'GotoAction' on the state stack and performing 'OnGoto' if set.
- DoReleaseTokens by removing child tokens from the nonterminal if 'StoreTokens' option is set to 'NoUserObject' or 'Never'

---

6. The Deserialization system.

The deserialization system allows parsers and lexers to be created from compiled grammar tables compacted to a file.
The CGTReader (stands for Compiled Grammar Table) gets filename or filestream as an input and returns a lexer or a parser with the corresponding rules. These rules are the DFA and the LALR and are the compiled grammar table.
Those rules are compacted in a CGTStructure that consists of Records and each record consists of Entries. Think of an entry as a single value that occupies a number of bytes in a file and of a record as a sequence of such Entries. The system is not some serialized .net object or anything but a plain old-school file structure that is read few bytes at a time.
The CGTContent are elements of the DFA and the LALR each represented by a different kind of record. The CGTStructure is translated to CGTContent structure from which the actual DFA and LALR classes are constructed.

CGTReader		
.(Stream)		
.(string)		
.CreateNewTokenizer()	***-> StringTokenizer***	*Creates DFA and lexer with it.*
.CreateNewParser()	***-> LALRParser***	*Creates DFA and lexer with it. Creates a new LALR structure and a new LALR parser with the lexer and the LALR structure.*

CGTStructure	
.string	***Header***
.Record[]	***Records***
.Entry[]	***Entries***

CGTContent	
Miscellaneous parameters of the compiled grammar.	
.Parameters	***Parameters***
.string	***Name***
.string	***Version***
.string	***Author***
.string	***About***
.bool	***CaseSensitive***
.int	***StartSymbol***

How many record there are for different things.		
.TableCounts	***TableCounts***	
.int	***SymbolTableCount***	
.int	***CharacterSetTableCount***	
.int	***RuleTableCount***	
.int	***DFATableCount***	
.int	***LALRTableCount***	
Collection of records that define symbols.		
.SymbolRecord[]	***SymbolTable***	
.int	***Index***	
.string	***Name***	
.int	***Kind***	
Collection of records that define character sets.		
.CharacterSetRecord[]	***CharacterSetTable***	
.int	***Index***	
.string	***Characters***	
Collection of records that define character rules to reduce tokens.		
.RuleRecord[]	***RuleTable***	
.int	***Index***	
.int	***Nonterminal***	
.int[]	***Symbols***	
The starting states for the DFA and LALR parser.		
.InitialStatesRecord	***InitialStatesRecord***	
.int	***DFA***	
.int	***LALR***	
Collection of records that define a DFA state.		
.DFAStateRecord[]	***DFAStateTable***	
.int	***Index***	
.bool	***AcceptState***	
.int	***AcceptIndex***	
.EdgeSubRecord[]	***EdgeSubRecords***	*The edges between DFA states.*
.int	***CharacterSetIndex***	
.int	***TargetIndex***	
.LALRStateRecord[]	***LALRStateTable***	
.int	***Index***	
.ActionSubRecord[]	***ActionSubRecords***	
.int	***SymbolIndex***	
.int	***Action***	
.int	***Target***	