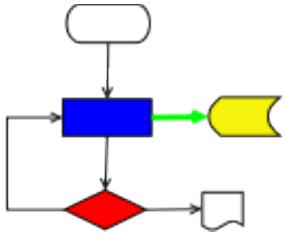
VICI



VISUAL CHART INTERPRETER

Design of libebnf

Publication History

Date	Who	What Changes
4 October 2012	Brenton Ross	Initial version.
25 May 2014	Brenton Ross	Updated for VICI
3 November 2014	Brenton Ross	Added increment plan references.

Table of Contents

1 Introduction

This is part of the system design document for the VICI project.

1.1 Scope

This document covers the design of libebnf, the parser for EBNF.

This design is for increment #3.

1.2 Overview

The detailed design includes:

- Interface Stubs: A framework of facade classes for the modules.
- Use Case Descriptions: A description of how a user is expected to interact with the application.
- Application Design: The classes and their relationships.
- User Interface Design: The design and layout of the graphical components of the system.
- Persistent Storage Design: The specifications for the XML files used to store configuration and scripts.

1.3 Audience

This document is intended to be used by the designers and developers, and later the maintainers, of the VICI project.

2 Responsibilities

This section outlines the responsibilities that the module must satisfy. This is followed by a discussion of the implications.

The architecture document lists T1.10 (Validate the EBNF of the option syntax) as the only responsibility that directly descends from a tactic, but goes on to mention that the library should also validate the options for a command against a provided EBNF and to provide a list of valid next symbols for a partial option list for a specific command.

The libsyntax library needs the EBNF in a portable form so that it can generate the syntax chart for a command.

The parsing operation will generally be preceded by a validation operation, so it makes sense for the library to retain the parse tree created by the validation routine, rather than having to go through the entire parsing again. It will need to determine if the parse request is for the same data as the previous validation, which can be done by retaining a copy of the string. (Using C++ STL strings this is a very small cost as they are reference counted.)

It is likely that the user will occasionally make errors. The validation routine should not treat these as exceptions – users are expected to make errors. The parser, however, will treat an error as an exception since the string should have been validated previously.

An interface function is defined in the interface that allows the user to determine location and nature of the problem. The validation routine will need to store the details of its errors so that this routine can fetch them.

2.1 Design Approach

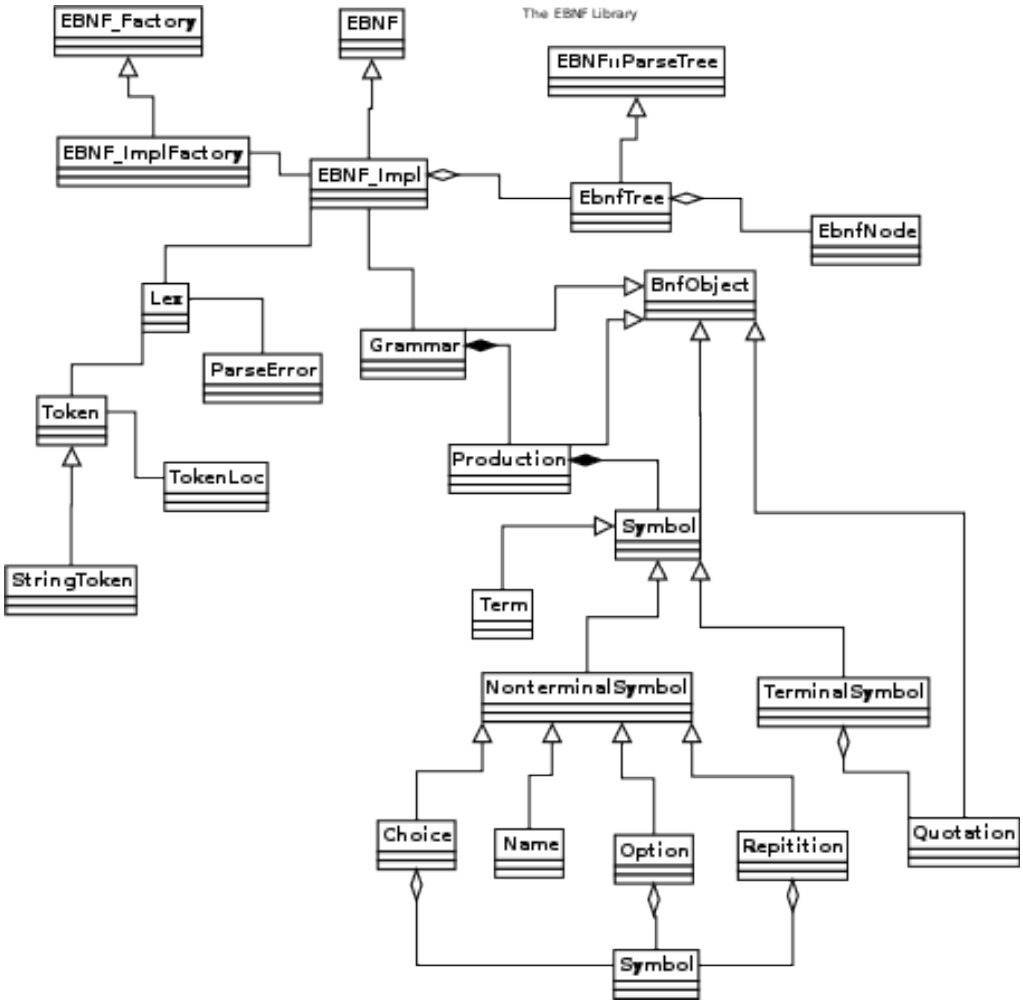
The library will require the candidate EBNF strings to be parsed, and hence some form of parser will be required. We could use lex/yacc or SableCC, but the language is quite simple, so a hand crafted recursive descent parser seems more appropriate.

A simple token parser, copied from another project, will get tokens from the string which will then be handed to the main parser. This code already contains useful features such as recording the position of the token, which is necessary if we intend to advise the user of the location of any errors we detect.

3 Library Design

The library consists of a lexical analyser that picks tokens from the source text and feeds them to a parser that constructs objects that correspond to the EBNF language.

The following diagram shows the static relationships between the classes:

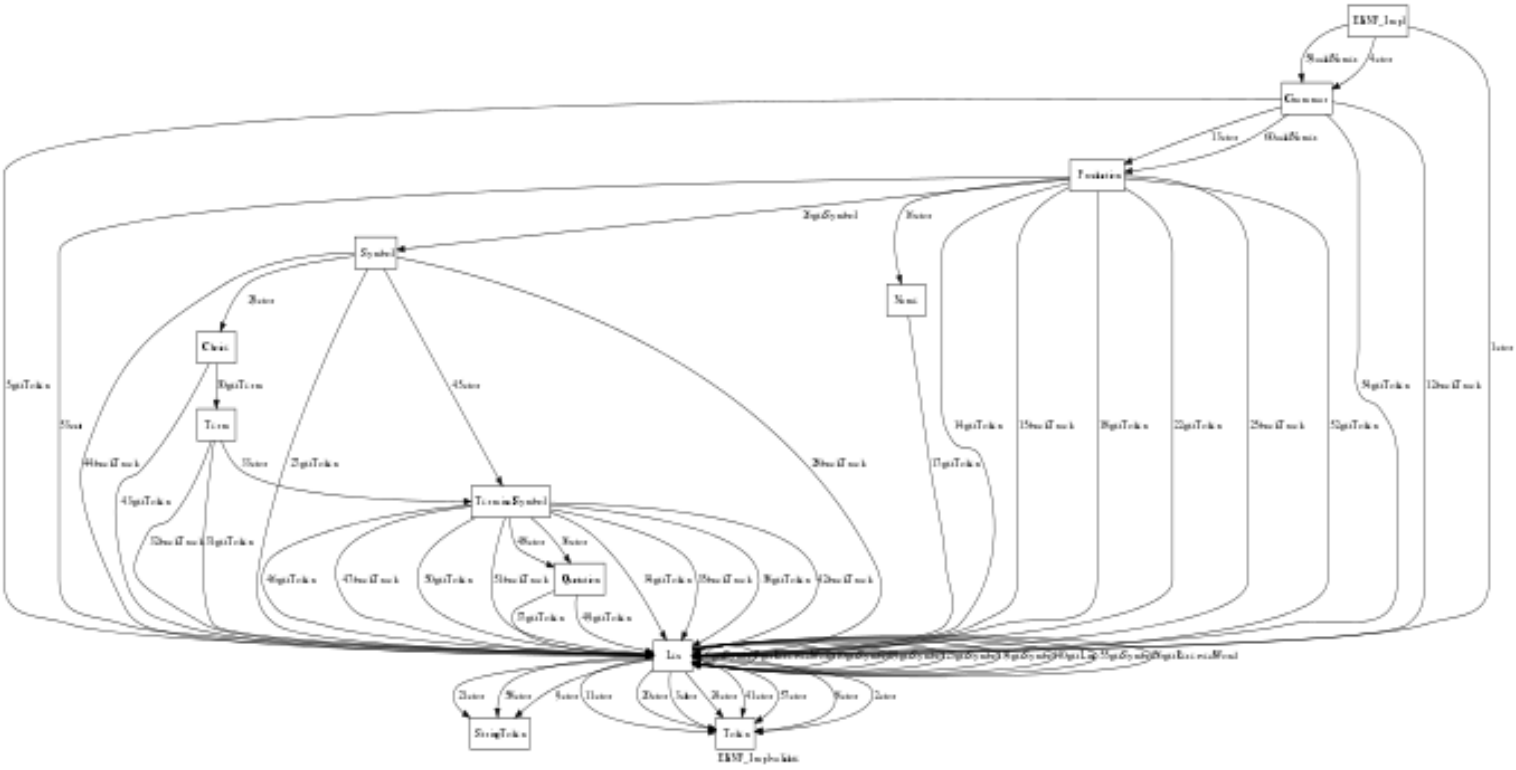


4 Collaboration Diagrams

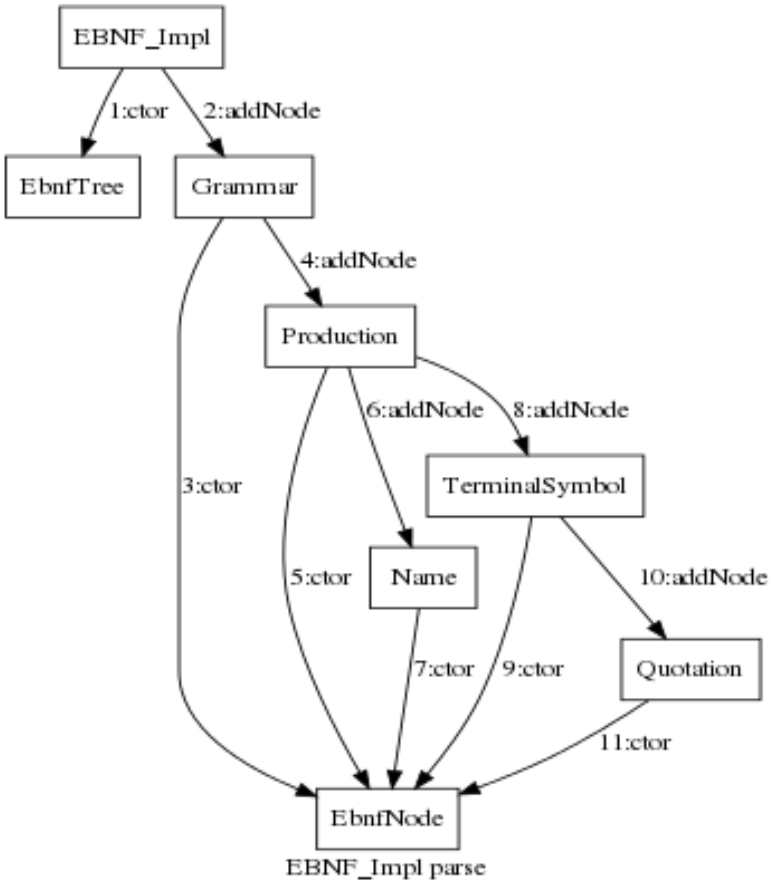
This section documents the interactions between the classes for each applicable use case.

The edges in the diagrams are numbered according to the sequence of events.

4.1 Validate



4.2 Parse



EBNF_Impl parse

5 Class Descriptions

This library will be responsible for parsing an EBNF specification for the options of a command.

5.1 EBNF_Factory Class

This provides an abstract interface that is used to create instances of the EBNF class. Different implementations of this class are used to create stubbed and test versions, as well as the production version. The FactoryFactory class implemented in libconfig is responsible for determining which concrete form is created.

```
class EBNF_Factory
{
public:
      virtual ~EBNF_Factory() {}
      virtual EBNF * makeEBNF() = 0;
};
```

5.2 EBNF_ImplFactory Class

This is a concrete instance of the EBNF_Factory class that creates the production version of the EBNF objects. (Other instances will create stubbed or test versions.)

```
class EBNF_ImplFactory : public EBNF_Factory
{
public:
      EBNF_ImplFactory() {}
      virtual EBNF * makeEBNF();
};
```

5.3 EBNF Class

This abstract class provides the facade for the library with the following responsibilities:

T1.10.1	Parse an EBNF string to confirm that its valid.
T1.10.2	Generate a list of valid next symbols, given the partial command option string.
T1.10.3	Create a parse tree for use by the syntax chart library.

```
class EBNF
{
public:
	virtual ~EBNF(){}

	// confirm that an EBNF is valid
	virtual bool validate( csr s ) = 0;

	// get location and details of parsing error
	virtual void getError( int & line, int & column, std::string &
text ) = 0;

	// parse the EBNF
	virtual ParseTree * parse( csr s ) = 0;
};
```

5.4 EBNF_Impl Class

This is a concrete instance of the EBNF class that provides the production implementation of the responsibilities.

```
class EBNF_Impl : public EBNF
{
public:
	EBNF_Impl();
	virtual bool validate( csr s );
	virtual void getError( int &line, int & column, std::string & message );
	virtual ParseTree * parse( csr s );
};
```

5.5 ParseTree Class

This abstract class provides a portable representation of the parse tree for a command. It is essentially used as a strongly typed void pointer so that the interface can be defined globally, but components that don't use the interface do not need an implementation.

```
class ParseTree
{
public:
	virtual ~ParseTree(){}
};
```

5.6 EbnfTree Class

This is the implementation of the ParseTree class. It is responsible for holding the root node of the parse tree.

```
class EbnfTree : public EBNF::ParseTree
{
public:
	EbnfTree();
	virtual ~EbnfTree();

	EbnfNode *root;
	void exportXml();
	void importXml();
};
```

5.7 EbnfNode Class

The entire parse tree is represented by a tree of these objects. Each one has pointers to its parent node (except the root), its next and previous siblings, as well as its first and last children. Each one contains the type of the node, and where applicable the text associated with the node.

```
class EbnfNode
{
public:
	enum NodeType { Undefined, Grammar, Production, Terminal, Quotation,
	                            Name, Repetition, Option, Choice };
	EbnfNode();
	~EbnfNode();

	EbnfNode *parent;
	EbnfNode *prev, *next;
	EbnfNode *firstChild, *lastChild;

	NodeType nodeType;
	std::string text;

	std::string typeOfNode() const;
	void typeOfNode( const std::string & );

};
```

5.8 Lex Class

This class is responsible for providing tokens to the parser for language elements. It is also responsible for keeping state information about where the parsing has got up to, and where errors are detected.

```
class Lex
{
public:
	/// The string values for the tokens
	static const char *tokens[];

	/** Constructor.
	@param fileName The file to parse.
	*/
	Lex( const char *fileName );

	/** Constructor
	@param text The string to parse.
	*/
	Lex( const std::string &text );

	/** Get next token.
	@return The next available token.
	*/
	Token *getToken();

	/** Backup so that next token is t.
	@param t The token that will be returned on the next call.
```

```
	*/
	void backTrack( Token *t );

	/** Cut off tokens that are no longer required.
	@param t The token that will be returned on the next call.
	*/
	void cut( Token *t );

	void showLocn( Token *t, std::ostream &s );
	ParseError &error();
};
```

5.9 ParseError Structure

This object is used to store the location and nature of errors so that they may be reported back to the user.

```
struct ParseError
{
	ParseError() : line(0), column(0) {}
	int line, column;
	std::string message;
};
```

5.10 TokenLoc Class

Used to store the location of a token.

```
class TokenLoc
{
public:
	int fileNum;		// index into an array of file names
	int lineNum;		// line number of that file
	int column;		// column number in that line

	// a default constructor so that these can be put in an array
	TokenLoc()
	{	fileNum = 1;
		lineNum = 1;
		column  = 0;
	}

	// normal constructor
	TokenLoc( int f, int L, int c )
	{	fileNum = f;
		lineNum = L;
		column = c;
	}
};
```

5.11 Token Class

The objects of this class hold one token of the lexical analysis process. This may be a reserved word or a symbol.

```
class Token
{
friend std::ostream& operator << ( std::ostream&, Token& );
public:
	enum TokenName {
	_ASSIGN_,
	_ELIPSIS_,
	_LEFT_BRACE_, _LEFT_SQUARE_BRACKET_,
	_PIPE_,
	_RIGHT_BRACE_, _RIGHT_SQUARE_BRACKET_,
	_SEMICOLON_, _String_,
	_Name_, _EndOfFile_, _Error_ };

	/// All tokens belong to one lex
	static Lex *lex;

	/// Identifies the token
	TokenName theToken;

	/// So we know where it came from in the source file
	TokenLoc posn;

	/// Reference to static token names
	const char *name;


	/** Constructor.
	<P> This constructor is called once just to set the static reference
	to the lexical analyser.
	@param lx The lexical analyser.
	*/
	Token( Lex *lx );

	/** Constructor.
	This constructor is called by the lexical analyser
	during the parsing process.
	@param n Enum representing the token found.
	@param p The position in the source files.
	*/
	Token( TokenName n, TokenLoc p );

	/// Increment the reference count.
	void incRefs();

	/// Decrement the reference count
	void decRefs();

	/// Check if there are any remaining references
	int  hasRefs();

	/// Destructor
	virtual ~Token();

	/// Equality operator
	virtual bool operator==( Token &t );

};
```

5.12 StringToken Class

This is a type of token which includes some text, such as a name or quoted string.

```
class StringToken : public Token
{
friend std::ostream& operator << ( std::ostream&, StringToken& );
public:
      /// The string value
      char *stringValue;

      /// Constructor
      StringToken( TokenName, TokenLoc p, const char *s );

      /// Destructor
      virtual ~StringToken();

      ///  Equality operator
      virtual bool operator==( Token &t );
};
```

5.13 BnfObject Class

This is an abstract base class for the parser objects used in the recursive descent parsing. It is responsible for those aspects which are common to all the parsing classes, such as maintaining a pointer to Lex object.

```
class BnfObject
{
protected:
      Lex & lex;
      bool ok;
public:
      BnfObject( Lex &L ) : lex(L), ok(false) {}
      virtual ~BnfObject() {}
      virtual void addNames( std::map< std::string, unsigned char > & names ) = 0;
      virtual EbnfNode * addNode( EbnfNode *parent ) = 0;
      bool isOK();
      void report( const TokenLoc &, const std::string & message );
};
```

5.14 Grammar Class

This class is responsible for recognising an entire EBNF. It provides the implementation for the EBNF facade.

T1.10.1	Parse an EBNF string to confirm that its valid.
T1.10.2	Generate a list of valid next symbols, given the partial command option string.
T1.10.3	Create a parse tree for use by the syntax chart library.

```
class Grammar : public BnfObject
{
protected:
	std::vector< Production * > productions;
public:
	Grammar( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

5.15 Production Class

This class is responsible for recognising a single production rule in the EBNF.

```
class Production : public BnfObject
{
protected:
	Name * name;
	std::vector< Symbol * > symbols;
public:
	Production( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );

};
```

5.16 Symbol Class

This class is responsible for recognising the symbols that make up the body of a production rule.

```
class Symbol : public BnfObject
{
public:
	Symbol( Lex & );
	static Symbol *getSymbol(Lex &);
};
```

5.17 TerminalSymbol Class

This class is responsible for recognising the actual strings used to form the options and parameters.

```
class TerminalSymbol : public Symbol
{
protected:
	Quotation * startQuote;
	Quotation * endQuote;		// may be null
public:
	TerminalSymbol( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

5.18 Quotation Class

This class is responsible for recognising quoted strings.

```
class Quotation : public BnfObject
{
protected:
	std::string text;
public:
	Quotation( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

5.19 NonterminalSymbol Class

This class is responsible for recognising the various constructs of the EBNF language.

```
class NonterminalSymbol : public Symbol
{
public:
	NonterminalSymbol( Lex & );
	static Symbol * getSymbol(Lex&);
};
```

5.20 Name Class

This class is responsible for recognising names within the EBNF.

```
class Name : public NonterminalSymbol
{
protected:
	std::string text;
public:
	Name( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
	std::string &name() { return text; }
};
```

5.21 Choice Class

This class is responsible for recognising choices between symbols.

```
class Choice : public NonterminalSymbol
{
protected:
	Symbol * left;
	Symbol * right;
public:
	Choice( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

5.22 Term Class

This class allows us to have multiple Choice classes by preventing the infinite recursion that would occur otherwise.

```
class Term : public Symbol
{
public:
	Term( Lex & );
	static Symbol *getTerm( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
};
```

5.23 Option Class

This class is responsible for recognising optional symbols.

```
class Option : public NonterminalSymbol
{
protected:
	std::vector< Symbol * > symbols;
public:
	Option( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

5.24 Repetition Class

This class is responsible for recognising repeating symbols.

```
class Repitition : public NonterminalSymbol
{
protected:
	std::vector< Symbol * > symbols;
public:
	Repitition( Lex & );
	virtual void addNames( std::map< std::string, unsigned char > & names );
	virtual EbnfNode * addNode( EbnfNode * );
};
```

Appendix A