<https://www.dreamincode.net/forums/topic/153718-fundamentals-of-parsing/> [1]

<http://www.goldparser.org/getting-started/1-languages.htm> [2]

<https://en.wikipedia.org/wiki/Parsing> [3]

<https://tomassetti.me/antlr-mega-tutorial/> [4]

<https://en.wikipedia.org/wiki/ANTLR> [5]

<https://en.wikipedia.org/wiki/Context-free_grammar> [6]

<https://blog.gopheracademy.com/advent-2017/parsing-with-antlr4-and-go/> [7]

<https://ivanyu.me/blog/2014/09/13/creating-a-simple-parser-with-antlr/>

[6]

Context-free grammar

A set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements.

A -> alpha

A -> beta

That means that A can be replaced with either alpha or beta.

In context-free grammar, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is always a nonterminal symbol.

[4]

There are three ways how to do a parser:

1. A regular expression
2. A parser generator
3. Writing own parser by hand

Regular Expression

- the lack of recursion, you can’t find an expression inside another one

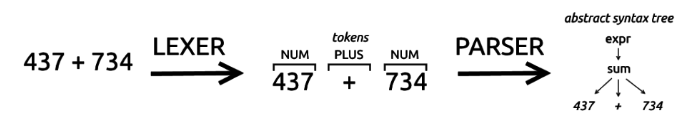
- it’s not really scalable

- hard to maintain

ANTLR vs writing own parser by hand

* The main advantage of ANTLR is productivity.
* Once to defined your grammars you can ask ANTLR to generate multiple parsers in different languages

Lexers and Parsers



[1]

Parsing of input into more easily processed components. Our input to the compiler is fundamentally free format providing we adhere to certain syntactic rules. In order to implement any parser, no matter how simple, you need to understand the input that you are required to write the parser for.

# Writing a Parser

We need to know exactly what tokens our parser is going to produce from the code we parse. A token represents a piece of the code and consists normally of two pieces of information, namely the token type and the token value. Some tokens can be described simply by their type (e.g. EOL).

## Look-ahead Mechanism

Parsers will generally need to provide a look-ahead mechanism which permits then to check to see if the token is complete. An example of this would be the sequence 0x12AB. This as we all know would need to be parsed as an integer not four separate tokens (0, x, 12, AB). To do this, if we read a zero as the lead character for an integer sequence, then we need to peek one character forwards to determine of the next character is a ‘X’ or ‘x’.

## Header Files

All tokens are derived from base token. It is the token itself that performs the parse of the source once the token has been identified.

The class TokenParser is simply the driving mechanism for the creation of the token classes. It’s task is to identify tokens based upon their leading character, create the correct token and the call that tokens ParseToken method.

[2]

A grammar is simply a formal definition of the syntax of a language. Every language has a formal syntax. When you describe the syntax of a language, you need to specify how each reserved word, identifier, etc. … will appear.

The term “syntax” refers to the structure of a programming language. The most common way of specifying the syntax of a language is use a notation known as Backus-Naur Form. BNF for short, is a notation to describe grammars. The notation breaks down the grammar into a series of rules – which are used to describe how the languages lexical and syntactic structures are used to form different logical units.

[3]

Parsing, syntax analysis or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information.

# Parser

A **parser** is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree of other hierarchical structure, giving a structural representation of the input while checking for correct syntax.

The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters.

## Lexical analyzer

**Lexical analysis**, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner. A lexer is generally combined with a parser, which together analyze the syntax of programming languages.

A lexical token or simply token is a string with an assigned and thus identified meaning. It is structured as a pair consisting of a token name and an optional taken value. The token name is a category of lexical unit. Common token names are:

* Identifier
* Keyword
* Separator (or punctuator)
* Operator
* Literal
* Comment

The specification of a language often includes a set of rules, the lexical grammar, which defines the lexical syntax. The lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions.

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters.

**Semantic analysis** is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking, or makes sure a variable is declared before use which is impossible to describe in the extended Backus-Naur form and thus not easily detected during parsing.

**Lexers** are often generated by a lexer generator. The most established is **lex**, paired with the **yacc** parser generator, and the free equivalents flex/bison.

## Parser 2

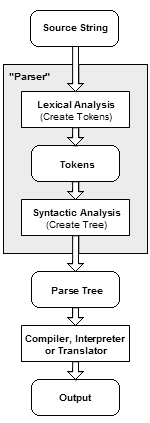
Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator.

## Overview of process

The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar.

The second stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar. However, not all rules defining programming languages can be expressed by context-free grammars alone, for example type validity and proper declaration of identifiers. These rules can be formally expressed with attribute grammars.

The third phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action.



## Types of parsers

The task of the parser is essentially to determine if and how the input can be derived from the start symbol of the grammar.

* Top-down parsing – LL parser
* Bottom-up parsing – LR parser

# ANTLR

It is a parser generator, a tool that helps you to create parsers. A parser takes a piece of text and transforms it in an organized structure, such as an Abstract Syntax Tree (AST). You can think of the AST as a story describing the content of the code.

What is needed:

1. Define a lexer and parser grammar
2. Invoke ANTLR, it will generate a lexer and a parser in your target language
3. Use the generated lexer and parser, you invoke them passing the code to recognize and they return to you and AST

# GOLD Parser

Lexical Token Structure

Nonterminal symbol

* Array of tokens
* Symbol

/// <summary>

/// Abstract class representing both terminal and nonterminal tokens.

/// </summary>

public abstract class Token

{

private Object userObject;

public Token()

{

this.userObject = null;

}

/// <summary>

/// This can be user for storing an object during the reduce

/// event. This makes it possible to create a tree when the

/// source is being parsed.

/// </summary>

public Object UserObject

{

get {return userObject;}

set {this.userObject = value;}

}

}

/// <summary>

/// Terminal token objects are retrieved from the tokenizer.

/// </summary>

public class TerminalToken : Token

{

private SymbolTerminal symbol;

private string text;

private Location location;

/// <summary>

/// Creates a new terminal token object.

/// </summary>

/// <param name="symbol">The symbol that this token represents.</param>

/// <param name="text">The text from the input that is the token.</param>

/// <param name="location">The location in the input that this token

/// has been found.</param>

public TerminalToken(SymbolTerminal symbol, string text, Location location)

{

this.symbol = symbol;

this.text = text;

this.location = location;

}

/// <summary>

/// String representation of the token.

/// </summary>

/// <returns></returns>

public override String ToString()

{

return text;

}

/// <summary>

/// The symbol that this token represents.

/// </summary>

public SymbolTerminal Symbol {get {return symbol;}}

/// <summary>

/// The text from the input that is this token.

/// </summary>

public string Text {get {return text;}}

/// <summary>

/// The location in the input that this token was found.

/// </summary>

public Location Location {get {return location;}}

}

/// <summary>

/// The nonterminal token is created when tokens are reduced by a rule.

/// </summary>

public class NonterminalToken : Token

{

private Token[] tokens;

private Rule rule;

/// <summary>

/// Creates a new nonterminal token.

/// </summary>

/// <param name="rule">The reduction rule.</param>

/// <param name="tokens">The tokens that are reduced.</param>

public NonterminalToken(Rule rule, Token[] tokens)

{

this.rule = rule;

this.tokens = tokens;

}

public void ClearTokens()

{

tokens = new Token[0];

}

/// <summary>

/// String representation of the nonterminal token.

/// </summary>

/// <returns>The string.</returns>

public override string ToString()

{

String str = rule.Lhs+" = [";

for (int i = 0; i < tokens.Length; i++)

{

str += tokens[i]+"]";

}

return str;

}

/// <summary>

/// The symbol that this nonterminal token represents.

/// </summary>

public SymbolNonterminal Symbol {get{return rule.Lhs;}}

/// <summary>

/// The tokens that are reduced.

/// </summary>

public Token[] Tokens {get{return tokens;}}

/// <summary>

/// The rule that caused the reduction.

/// </summary>

public Rule Rule {get{return rule;}}

}

/// <summary>

/// SymbolNonterminal is for symbols that are not directly linked to one token.

/// </summary>

public class SymbolNonterminal : Symbol

{

public SymbolNonterminal(int id, string name) : base(id,name)

{

}

public override String ToString()

{

return "<"+base.ToString()+">";

}

}

/// <summary>

/// SymbolTerminal is a symbol that is linked to a token.

/// </summary>

public class SymbolTerminal : Symbol

{

public SymbolTerminal(int id, string name) : base(id,name)

{

}

}

## C++ DLL

Unlike a statically linked library, Windows connects the imports in your app to the exports in a DLL at load time or at run time, instead of connecting them at link time. Windows requires extra information that isn't part of the standard C++ compilation model to make these connections. The Visual C++ compiler implements some Microsoft-specific extensions to C++ to provide this extra information. We explain these extensions as we go.

The DLL uses the C calling convention so it can be called from apps built by using other languages, as long as the platform and calling and linking conventions match. The client app uses implicit linking, where Windows links the app to the DLL at load-time. This lets the app call the DLL-supplied functions just like the functions in a statically linked library.

<https://docs.microsoft.com/en-us/cpp/build/dlls-in-visual-cpp>

In Windows, a dynamic-link library (DLL) is a kind of executable file that acts as a shared library of functions and resources. Dynamic linking is an operating system capability that enables an executable to call functions or use resources stored in a separate file. These functions and resources can be compiled and deployed separately from the executables that use them. A DLL is not a stand-alone executable; it runs in the context of an application that calls it. The operating system can load the DLL into an application's memory space when the application is loaded (implicit linking), or on demand at runtime (explicit linking). DLLs also make it easy to share functions and resources across executables. Multiple applications can access the contents of a single copy of a DLL in memory at the same time.

Exporting from the DLL:

A DLL file has a layout very similar to an .exe file, with one important difference — a DLL file contains an exports table. The exports table contains the name of every function that the DLL exports to other executables. These functions are the entry points into the DLL; only the functions in the exports table can be accessed by other executables. Any other functions in the DLL are private to the DLL. The exports table of a DLL can be viewed by using the [DUMPBIN](https://docs.microsoft.com/en-us/cpp/build/reference/dumpbin-reference) tool with the /EXPORTS option.

This walkthrough creates a DLL that can only be called from apps that use C++ calling conventions.

<https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2012/dt232c9t%28v%3dvs.110%29>

For Visual Basic applications (or applications in other languages such as Pascal or Fortran) to call functions in a C/C++ DLL, the functions must be exported using the correct calling convention without any name decoration done by the compiler.

\_\_stdcall creates the correct calling convention for the function (the called function cleans up the stack and parameters are passed from right to left) but decorates the function name differently. So, when **\_\_declspec(dllexport)** is used on an exported function in a DLL, the decorated name is exported.

The \_\_stdcall name decoration prefixes the symbol name with an underscore (\_) and appends the symbol with an at sign (@) character followed by the number of bytes in the argument list (the required stack space). As a result, the function when declared as:

int \_\_stdcall func (int a, double b)

is decorated as:

\_func@12

Because there is no way to override where the stack cleanup occurs, you must use \_\_stdcall. To undecorate names with \_\_stdcall, you must specify them by using aliases in the EXPORTS section of the .def file.

Pull Parsing versus Push Parsing

Streaming pull parsing refers to a programming model in which a client application calls methods on an parsing library when it needs.

Streaming push parsing refers to a programming model in which an parser sends (pushes) data to the client as the parser encounters elements in an data. The parser sends the data whether or not the client is ready to use it at the time.