http://www.kiwwito.com/article/build-a-lexical-analyzer-with-javacc

15.04.10 - Build a lexical analyzer with JavaCC

How to create a lexical analyzer with JavaCC for a defined programming language

In programming, a lexical analyzer is the part of a compiler or a parser that break the input language into tokens.

A token is the minimal meaning component. Common tokens are **identifiers**, **integers**, **floats**, **constants**, etc.

For building it, we are going to use an incredible useful tool, **JavaCC**. With simple **regular expressions** we can define our language tokens.

About JavaCC:

JavaCC is a tool usually used for parsers and is actually maintained by Sun Microsystems. Is very simple, efficient and safe. You can freely download it from the official website (https://javacc.dev.java.net/) or like a Eclipse plugin.

Practical case, build a lexical analyzer for a determined defined language:

Our language specifications are:
Tokens:

- **Constants**:
  - **Strings**: Characters between quotes, example: "cadena"
  - **Integers**: Positive numbers, example: 234 or 0
  - **Logicals**: TRUE and FALSE
- **Identifiers**: All the identifiers are a sequence of letters (a-zA-Z) and numbers that must start with a letter (and not a number). The **identifiers that refers to strings must end with a dollar** ($).
- **Reserved words**: In the language are some reserved words that refers to programming structures that brings to life the language. Those are "**not, if, end, let, call, then, case, else, input, print, select, and static**".
- Also, the language is **case insensitive**, that is, an identifier named "id" refers to the same point that another called "Id", "iD" or "ID". The same methodology for reserved words.

JavaCC code (exparser.jj):

```
options {
  IGNORE_CASE = true;
}
PARSER_BEGIN(ExampleParser)

  public class ExampleParser {
```

```
  //Parser execution
  public static void main ( String args [ ] ) {

    //Parser initialization
    ExampleParser parser;

    if(args.length == 0){
      System.out.println ("ExampleParser: Reading input ...");
      parser = new ExampleParser(System.in);
    }
    else if(args.length == 1){
      System.out.println ("ExampleParser: Reading the file " + args[0] + " ..." );
      try {
        parser = new ExampleParser(new java.io.FileInputStream(args[0]));
      }
      catch(java.io.FileNotFoundException e) {
        System.out.println ("ExampleParser: The file " + args[0] + " was not found.");
        return;
      }
    }
    else {
      System.out.println ("ExampleParser:  You must use one of the following:");
      System.out.println ("         java ExampleParser < file");
      System.out.println ("Or");
      System.out.println ("         java ExampleParser file");
      return ;
    }
    try {
      compilador.Start();
      System.out.println ("ExampleParser: The input was readed sucessfully.");
    }
    catch(ParseException e){
      System.out.println ("ExampleParser: There was an error during the parse.");
      System.out.println (e.getMessage());
    }
    catch(TokenMgrError e){
      System.out.println ("ExampleParser: There was an error.");
      System.out.println (e.getMessage());
    }
  }
 }
PARSER_END(ExampleParser)

//STRUCTURES AND CHARACTERS TO SCAPE
SKIP : {
 " "
|  "\t"
|  "\n"
|  "\r"
|  <"rem" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
}
//STATIC TOKENS
TOKEN : {
 <INTEGER_CONSTANT: (<DIGIT>)+>
|  <LOGIC_CONSTANT: "true" | "false" | "-1">
|  <STRING_CONSTANT: "\"" ( ~["\"","\\","\n","\r"] | "\\" ( ["n","t","b","r","f","\\","\'","\""] | (
["\n","\r"] | "\r\n")))* "\"">
|  <#DIGIT: ["0"-"9"]>
}
//RESERVED WORDS
TOKEN : {
 <NOT: "not">
|  <IF: "if">
|  <END: "end">
|  <SUB: "sub">
|  <LET: "let">
|  <CALL: "call">
|  <THEN: "then">
|  <CASE: "case">
|  <ELSE: "else">
|  <INPUT: "input">
|  <PRINT: "print">
|  <SELECT: "select">
|  <STATIC: "static">
}
//IDENTIFIER TOKEN
TOKEN : {
 <IDENTIFIER: <LETTER>(<LETTER>|<DIGIT>)*(["$"])?>
|  <#LETTER: (["a"-"z","A"-"Z"])>
}
//MAIN UNIT
void Start () : {}
{
  (
   INTEGER_CONSTANT | STRING_CONSTANT | LOGIC_CONSTANT |
   NOT | IF | END | SUB | LET | CALL | THEN | CASE | ELSE | INPUT | PRINT | SELECT | STATIC |
   IDENTIFIER
  )*
  <EOF>
}
```

For compiling this file you have to use "**javacc**" and after "**javac**":

```
$ javacc exparser.jj
```

```
$ javac *.java
```

For executing the program:

```
$ java ExampleParser file
```

Remember that the previous code is for the lexical analyzer, it not try to check the programming structures or compiling it to binary code, it only break the input into tokens. However, you can follow the development and make a compiler or a parser.

JavaCC [tm]: Command Line Syntax

First, you can obtain a synopsis of the command line syntax by simply typing "javacc". This is what you get:

```
% javacc
<<<< Version and copyright info>>>

Usage:
    javacc option-settings inputfile

"option-settings" is a sequence of settings separated by
spaces.
Each option setting must be of one of the following forms:

    -optionname=value (e.g., -STATIC=false)
    -optionname:value (e.g., -STATIC:false)
    -optionname       (equivalent to -optionname=true.
e.g., -STATIC)
    -NOoptionname     (equivalent to -optionname=false.
e.g., -NOSTATIC)

Option settings are not case-sensitive, so one can say "-
nOsTaTiC" instead
```

```
of "-NOSTATIC".  Option values must be appropriate for the
corresponding option, and must be either an integer, a
boolean, or a string value.

The integer valued options are:

    LOOKAHEAD                (default 1)
    CHOICE_AMBIGUITY_CHECK   (default 2)
    OTHER_AMBIGUITY_CHECK    (default 1)

The boolean valued options are:

    STATIC                   (default true)
    SUPPORT_CLASS_VISIBILITY_PUBLIC (default true)
    DEBUG_PARSER             (default false)
    DEBUG_LOOKAHEAD          (default false)
    DEBUG_TOKEN_MANAGER      (default false)
    ERROR_REPORTING          (default true)
    JAVA_UNICODE_ESCAPE      (default false)
    UNICODE_INPUT            (default false)
    IGNORE_CASE              (default false)
    COMMON_TOKEN_ACTION      (default false)
    USER_TOKEN_MANAGER       (default false)
    USER_CHAR_STREAM         (default false)
    BUILD_PARSER             (default true)
    BUILD_TOKEN_MANAGER      (default true)
    TOKEN_MANAGER_USES_PARSER (default false)
    SANITY_CHECK             (default true)
    FORCE_LA_CHECK           (default false)
    CACHE_TOKENS             (default false)
    KEEP_LINE_COLUMN         (default true)

The string valued options are:
    OUTPUT_DIRECTORY         (default Current Directory)
    TOKEN_EXTENDS            (java.lang.Object)
    TOKEN_FACTORY            (java.lang.Object)
    JDK_VERSION              (1.5)
    GRAMMAR_ENCODING         (default file.encoding)
```

```
EXAMPLE:
    javacc -STATIC=false -LOOKAHEAD:2 -debug_parser
mygrammar.jj

ABOUT JavaCC:

    JavaCC is a parser generator for the Java [tm]
programming
    language originally built by
    Sriram Sankar (http://www.cs.stanford.edu/~sankar) and
    Sreeni Viswanadha (http://www.cs.albany.edu/~sreeni).

%
```

- **LOOKAHEAD:** The number of tokens to look ahead before making a decision at a choice point during parsing. The default value is 1. The smaller this number, the faster the parser. This number may be overridden for specific productions within the grammar as described later. See the description of the lookahead algorithm for complete details on how lookahead works.
- **CHOICE_AMBIGUITY_CHECK:** This is an integer option whose default value is 2. This is the number of tokens considered in checking choices of the form "A | B | ..." for ambiguity. For example, if there is a common two token prefix for both A and B, but no common three token prefix, (assume this option is set to 3) then JavaCC can tell you to use a lookahead of 3 for disambiguation purposes. And if A and B have a common three token prefix, then JavaCC only tell you that you need to have a lookahead of 3 *or more*. Increasing this can give you more comprehensive ambiguity information at the cost of more processing time. For large grammars such as the Java grammar, increasing this number any further causes the checking to take too much time.
- **OTHER_AMBIGUITY_CHECK:** This is an integer option whose default value is 1. This is the number of tokens considered in checking all other kinds of choices (i.e., of the forms "(A)*", "(A)+", and "(A)?") for ambiguity. This takes more time to do than the choice checking, and hence the default value is set to 1 rather than 2.
- **STATIC:** This is a boolean option whose default value is true. If true, all methods and class variables are specified as static in the generated parser

and token manager. This allows only one parser object to be present, but it improves the performance of the parser. To perform multiple parses during one run of your Java program, you will have to call the ReInit() method to reinitialize your parser if it is static. If the parser is non-static, you may use the "new" operator to construct as many parsers as you wish. These can all be used simultaneously from different threads.

- **DEBUG_PARSER:** This is a boolean option whose default value is false. This option is used to obtain debugging information from the generated parser. Setting this option to true causes the parser to generate a trace of its actions. Tracing may be disabled by calling the method disable_tracing() in the generated parser class. Tracing may be subsequently enabled by calling the method enable_tracing() in the generated parser class.
- **DEBUG_LOOKAHEAD:** This is a boolean option whose default value is false. Setting this option to true causes the parser to generate all the tracing information it does when the option DEBUG_PARSER is true, and in addition, also causes it to generated a trace of actions performed during lookahead operation.
- **DEBUG_TOKEN_MANAGER:** This is a boolean option whose default value is false. This option is used to obtain debugging information from the generated token manager. Setting this option to true causes the token manager to generate a trace of its actions. This trace is rather large and should only be used when you have a lexical error that has been reported to you and you cannot understand why. Typically, in this situation, you can determine the problem by looking at the last few lines of this trace.
- **ERROR_REPORTING:** This is a boolean option whose default value is true. Setting it to false causes errors due to parse errors to be reported in somewhat less detail. The only reason to set this option to false is to improve performance.
- **JAVA_UNICODE_ESCAPE:** This is a boolean option whose default value is false. When set to true, the generated parser uses an input stream object that processes Java Unicode escapes (\u...) before sending characters to the token manager. By default, Java Unicode escapes are not processed. This option is ignored if either of options USER_TOKEN_MANAGER, USER_CHAR_STREAM is set to true.
- **UNICODE_INPUT:** This is a boolean option whose default value is false. When set to true, the generated parser uses uses an input stream object that reads Unicode files. By default, ASCII files are assumed.
  This option is ignored if either of options USER_TOKEN_MANAGER, USER_CHAR_STREAM is set to true.

- **IGNORE_CASE:** This is a boolean option whose default value is false. Setting this option to true causes the generated token manager to ignore case in the token specifications and the input files. This is useful for writing grammars for languages such as HTML. It is also possible to localize the effect of IGNORE_CASE by using an alternate mechanism described later.
- **USER_TOKEN_MANAGER:** This is a boolean option whose default value is false. The default action is to generate a token manager that works on the specified grammar tokens. If this option is set to true, then the parser is generated to accept tokens from any token manager of type "TokenManager" - this interface is generated into the generated parser directory.
- **SUPPORT_CLASS_VISIBILITY_PUBLIC:** This is a boolean option whose default value is true. The default action is to generate support classes (such as Token.java, ParseException.java etc) with *Public* visibility. If set to false, the classes will be generated with package-private visibility.
- **USER_CHAR_STREAM:** This is a boolean option whose default value is false. The default action is to generate a character stream reader as specified by the options JAVA_UNICODE_ESCAPE and UNICODE_INPUT. The generated token manager receives characters from this stream reader. If this option is set to true, then the token manager is generated to read characters from any character stream reader of type "CharStream.java". This file is generated into the generated parser directory.
  This option is ignored if USER_TOKEN_MANAGER is set to true.
- **BUILD_PARSER:** This is a boolean option whose default value is true. The default action is to generate the parser file ("MyParser.java" in the above example). When set to false, the parser file is not generated. Typically, this option is set to false when you wish to generate only the token manager and use it without the associated parser.
- **BUILD_TOKEN_MANAGER:** This is a boolean option whose default value is true. The default action is to generate the token manager file ("MyParserTokenManager.java" in the above example). When set to false the token manager file is not generated. The only reason to set this option to false is to save some time during parser generation when you fix problems in the parser part of the grammar file and leave the lexical specifications untouched.
- **TOKEN_MANAGER_USES_PARSER:** This is a boolean option whose default value is false. When set to true, the generated token manager will include a field called `parser` that references the instantiating parser instance (of type `MyParser` in the above example). The main reason for

having a parser in a token manager is using some of its logic in lexical actions. This option has no effect if the STATIC option is set to true.

- **TOKEN_EXTENDS:** This is a string option whose default value is "", meaning that the generated Token class will extend java.lang.Object. This option may be set to the name of a class that will be used as the base class for the generated `Token` class.
- **TOKEN_FACTORY:** This is a string option whose default value is "", meaning that Tokens will be created by calling `Token.newToken()`. If set the option names a Token factory class containing a `public static Token newToken(int ofKind, String image)` method.
- **SANITY_CHECK:** This is a boolean option whose default value is true. JavaCC performs many syntactic and semantic checks on the grammar file during parser generation. Some checks such as detection of left recursion, detection of ambiguity, and bad usage of empty expansions may be suppressed for faster parser generation by setting this option to false. Note that the presence of these errors (even if they are not detected and reported by setting this option to false) can cause unexpected behavior from the generated parser.
- **FORCE_LA_CHECK:** This is a boolean option whose default value is false. This option setting controls lookahead ambiguity checking performed by JavaCC. By default (when this option is false), lookahead ambiguity checking is performed for all choice points where the default lookahead of 1 is used. Lookahead ambiguity checking is not performed at choice points where there is an explicit lookahead specification, or if the option LOOKAHEAD is set to something other than 1. Setting this option to true performs lookahead ambiguity checking at *all* choice points regardless of the lookahead specifications in the grammar file.
- **COMMON_TOKEN_ACTION:** This is a boolean option whose default value is false. When set to true, every call to the token manager's method "getNextToken" (see the description of the Java Compiler Compiler API) will cause a call to a used defined method "CommonTokenAction" after the token has been scanned in by the token manager. The user must define this method within the TOKEN_MGR_DECLS section. The signature of this method is:
- `    void CommonTokenAction(Token t)`
- **CACHE_TOKENS:** This is a boolean option whose default value is false. Setting this option to true causes the generated parser to lookahead for extra tokens ahead of time. This facilitates some performance improvements. However, in this case (when the option is true), interactive

applications may not work since the parser needs to work synchronously with the availability of tokens from the input stream. In such cases, it's best to leave this option at its default value.

- **OUTPUT_DIRECTORY:** This is a string valued option whose default value is the current directory. This controls where output files are generated.