ANTLR

Ashley J.S Mills

<ashley@ashleymills.com>
Copyright © 2005 The University Of Birmingham

Table of Contents

1. Introduction

ANTLR (ANother Tool for Language Recognition) is a parser and translator generator tool that lets one define language grammars in either ANTLR syntax (which is YACC and EBNF(Extended Backus-Naur Form) like) or a special AST(Abstract Syntax Tree) syntax. ANTLR can create lexers, parsers and AST's. ANTLR is more than just a grammar definition language however, the tools provided allow one to implement the ANTLR defined grammar by automatically generating lexers and parsers (and tree parsers) in either Java (http://java.sun.com/, C++ (http://anubis.dkuug.dk/jtc1/sc22/wg21/ or Sather (http://www.icsi.berkeley.edu/~sather/.

ANTLR implements a PRED-LL(k) parsing strategy and affords arbitrary lookahead for disambiguating the ambiguous. An answer to the question "What is ANTLR?" by Terrance Parr the creator of ANTLR can be found here: http://www.jguru.com/faq/view.jsp?EID=77

2. Background Information

ANTLR is a compiler tool hence it's developer base is generally constrained to those whom desire to create translators of some kind. In order to comprehend much of what will be discussed in this tutorial it is necessary to first get a feel of the terminology used in this area of computer science and the basic concepts behind the operation of ANTLR. This section will begin with a brief discussion of how a compiler operates.

2.1. The Lexer

Other names: Scanner, lexical analyser, tokenizer.

Programming languages are made up of keywords, and strictly defined constructs, the ultimate aim of the compilation process is to translate the high level instructions of the programming language into the low-level instructions of the machine or virtual machine that is the intended execution architecture. For example, a native C++ compiler compiles C++ code into machine language instructions that execute directly on the target hardware (or on some simulation of the target hardware), the standard Java compiler distributed by Sun Microsystems compiles Java source code to Java bytecode which is the machine language instruction set used by the Java virtual machine, this bytecode can then be executed by any platform that implements the Java virtual machine.

A source program is written using some kind of editing tool that can produce a file which is comprised of statements and constructs that are allowed in the programming language being used. The actual text of the file is written using characters of a particular character set or subset of some character set, so a source file can be thought of as a stream of characters terminated by some EOF (End Of File) marker that signifies the end of the source file.

A source file is streamed to a lexer on a character by character basis by some kind of input interface. The lexers job is to quantify

the meaningless stream of characters into discrete groups that, when processed by the parser, have meaning. Each character or group of characters quantified in this manner is called a token. Tokens are components of the programming language in question such as keywords, identifiers, symbols, and operators. (Usually)The lexer removes comments and whitespace from the program, and any other content that is not of semantic value to the interpretation of the program. The lexer converts the stream of characters into a stream of tokens which have individual meaning as dictated by the lexer's rules. Similarly, your brain is probably grouping the individual characters that make up each of the words in this sentence into tokens (words in this case, which have semantic value to you), your job of determining where one token finishes and another begins is made a little easier however, because the words in a sentence are already separated by spaces, it could be argued that an English sentence is already tokenised in this sense, however, we can assume that some kind of grouping and recognition is occurring at the word level too. The stream of tokens generated by the lexer are received by the parser.

A lexer usually generates errors pertaining to sequences of characters it cannot match to a specific token type defined by one of it's rules.

2.2. The Parser

Other Names: Syntactical analyser.

A lexer groups sequences of characters it recognises in the character stream into tokens with individual semantic worth, it does not consider their semantic worth within the context of the whole program, this is the job of the parser. Languages are described by a grammar, the grammar determines exactly what defines a particular token and what sequences of tokens are decreed as valid. The parser organises the tokens it receives into the allowed sequences defined by the grammar of the language. If the language is being used exactly as is defined in the grammar, the parser will be able to recognise the patterns that make up certain structures and group these together. If the parser encounters a sequence of tokens that match none of the allowed sequences of tokens, it will issue an error and perhaps try to recover from the error by making a few assumptions about what the error was.

The parser checks to see if the tokens conform to the syntax of the language defined by the grammar. Similarly your brain knows what kinds of sentences are valid within a particular language such as English and it could be said that at this moment in time your brain is parsing the words of this sentence and grouping them into what you understand as valid sequences, for instance, your brain knows that a sentence ends when a full stop is encountered, one would not assume that the text following the full stop was part of the same sentence. In addition to this your brain is also extracting meaningful information from the sentence. Usually the parser will convert the sequences of tokens that it has been deliberately created to match into some other form such as an Abstract Syntax Tree (AST). An AST is easier to translate to a target language because an AST contains additional information implicitly, by nature of it's structure. Effectively, creating an AST is the most important part of a language translation process.

The parser generates one or more symbol table(s) which contain information regarding tokens it encounters, such as whether or not the token is the name of a procedure or if it had some specific value, the symbol tables are used in the generation of object code and in type checking, for example, so that an integer cannot be assigned to a string or whatever. ANTLR uses symbol tables to speed up the matching of tokens, in that an integer is mapped to a particular token, then instead of matching the string that would compose a textual description of that token, the integer that represents that token is matched instead, which is a lot quicker. Eventually the AST will be translated to an executable format, some linking of libraries may be performed, this is not considered the job of the compiler and is not of direct concern here.

A parser usually generates errors pertaining to sequences of tokens it cannot match to the specific syntactical arrangements allowed, as decreed by the grammar.

Both lexers and parsers are recognizers, lexers recognize sequences of characters, parsers recognize sequences of Tokens. A lexer or a parser converts a stream of elements (be they characters or tokens) and translates them to some other stream of elements such as tokens representing larger structures or groups of elements or perhaps nodes in an abstract syntax tree. They are essentially the same thing, however, traditionally lexers are associated with processing streams of characters and parsers are associated with processing streams of Tokens.

It is recommended that you read *Building Recognizers By Hand* by Terrance Parr the creator of ANTLR, it can be found here http://www.antlr.org/book/byhand.pdf, to get an insight into how one would go about creating a recogniser in Java, and from this you can abstract how it can be done in any programming language. When you have the time you should read all the documentation that came with the ANTLR installation.

2.3. What is ANTLR's part in all this?

ANTLR lets you define the rules that the lexer should use to tokenize a stream of characters and the rules the parser should use to interpret a stream of tokens. ANTLR can then generate a lexer and a parser which you can use to interpret programs written in your language and translate them other languages and AST's. The design of ANTLR affords much extensibility and it has many applications.

3. Installation

The documentation for the installation is written under the assumption that the reader has some experience of installing software on computers and knows how to change the operating environment of the particular operating system they are using. The documents entitled *Configuring A Windows Working Environment* [../winenvars/winenvarshome.html] and *Configuring A Unix Working Environment* [../unixenvars/unixenvarshome.html] are of use to people who need to know more.

1. Obtain the ANTLR download by following the download section links at http://www.antlr.org/

2. Unzip to a suitable location.

3. Add, `/path/to/where/you/unzipped/ANTLR/antlr.jar` and `/path/to/where/you/unzipped/ANTLR` to your classpath, do not include a trailing slash after the directory name otherwise you may encounter problems.

4. ANTLR Grammar Template

An ANTLR grammar file has a number of components, some of which are optional and some of which are mandatory, the 'template' below shows all the components that make up an ANTLR grammar file and then briefly describes them, do not expect to comprehend this instantly, things will become clearer the further you progress into this document. And then you may use this template to remind yourself what is allowed where within an ANTLR grammar.

```
header {
  // stuff that is placed at the top of <all> generated files
}       ❶

options { options for entire grammar file }

{ optional class preamble - output to generated classfile
immediately before the definition of the class }
class YourLexerClass extends Lexer;
// definition extends from here to next class definition
// (or EOF if no more class defs)
options { YourOptions }
tokens...❷
lexer rules...
myrule[args] returns [retval]
   options { defaultErrorHandler=false; }
   :   // body of rule...
   ;

{ optional class preamble - output to generated classfile
immediately before the definition of the class }
class YourParserClass extends Parser;
options { YourOptions }
tokens...
parser rules...

{ optional class preamble - output to generated classfile
immediately before the definition of the class }
class YourTreeParserClass extends TreeParser;
options { YourOptions }
tokens...
tree parser rules...

// arbitrary lexers, parsers and treeparsers may be included
```

❶

```
header {
  // stuff that is placed at the top of <all> generated files
}
```

The textual content of the header will be copied verbatim to the top of all files generated when ANTLR is ran on the grammar.

❷

```
{ optional class preamble - output to generated classfile
          immediately before the definition of the class }
class YourLexerClass extends Lexer;
// definition extends from here to next class definition
// (or EOF if no more class defs)
options { YourOptions }
tokens...
lexer rules...
```

This begins with an optional class preamble, any text placed here will be copied verbatim to the top of the class the statement prefixes. The options section contains options specific to this class, for example:

```
options {
   k = 2;                        // Set token lookahead to two
```

```
    tokenVocabulary = Befunge;   // Call it's vocabulary "Befunge"
    defaultErrorHandler = false; // Don't generate parser error handlers
}
```

Extensive detail pertaining to the various options available can be found in the ANTLR documentation that is installed with everything else, i.e `/path/to/where/you/installed/ANTLR/doc/options`. The tokens section lets you explicitly define literals and imaginary tokens, e.g:

```
tokens {
   EXPR;          // Imaginary token
   THIS="that";   // Literal definition
   INT="int";     // Literal definition
}
```

The lexer rules come next and have the general form:

```
rulename [args] returns [retval]
   options { defaultErrorHandler=false; }
   { optional initiation code }
   :   alternative_1
   |   alternative_2
   ...
   |   alternative_n
   ;
```

For example:

```
INT    : ('0'..'9')+; // Matches an integer
```

There can be an arbitrary number of rules.

The other two sections have the same layout as the one described. There can be zero or more lexers, parsers and treeparsers and they can come in any order. The scope of the a class is defined as extending from the class's declaration to the next class declaration, or if it is the last class declaration, to the end of the file.

5. ANTLR Notation

ANTLR specifies it's lexical rules and parser rules using the almost exactly the same notation, ANTLR notation is based on YACC's notation and there are some EBNF constructs thrown in for good measure. A rule is simply a sequence of instructions which describe a particular pattern that ANTLR should match.

5.1. Zero Or More

ANTLR uses the notation *(expression)** to indicate that the pattern matching expression specified inside the parentheses must be matched zero or more times.

5.2. One Or More

ANTLR uses the notation *(expression)+* to indicate that the pattern matching expression specified inside the parentheses must be matched one or more times.

5.3. Optional

ANTLR uses the notation *(expression)?* to indicate that the pattern matching expression specified inside the parentheses must be matched zero or one times, in other words, it's optional.

6. Lexer Example

This example illustrates a very simple lexer that matches alpha and numeric strings. It is available to download here: simple.g [files/simple.g].

```
class SimpleLexer extends Lexer;

options { k=1; filter=true; }

ALPHA   : ('a'..'z'|'A'..'Z')+
        { System.out.println("Found alpha: "+getText()); }
        ;

NUMERIC : ('0'..'9')+
        { System.out.println("Found numeric: "+getText()); }
        ;

EXIT    : '.' { System.exit(0); } ;
```

This will be explained in sections, lets begin with the first line:

```
class SimpleLexer extends Lexer;
```

This is the lexer declaration, pretty straightforward, it's scope is from the line shown to the next class declaration or, if there are no more class declarations, until the end of the file.

```
options { k=1; filter=true; }
```

Here some basic options are set. *k* is set to one, *k* is the lookahead value. For example, with a lookahead of one, ANTLR would not be able to tell the difference between:

```
      SILLY1 : "ab" ;
      SILLY2 : "ac" ;
```

And when trying to parse a file containing these lexer rules, ANTLR will issue the error:

```
warning: lexical nondeterminism between rules SILLY1 and SILLY2 upon
silly.g:0:  k==1:'a'
```

Because if the lexer encountered "ab", with a lookahead of one, it would get confused as to whether it should match the rule SILLY1 or the rule SILLY2 since they both begin with 'a'. With k=2, that is, a lookahead of two, the lexer will not only compare the first character but also the second character and hence will be able to disambiguate between the two cases. There are cases when increasing the lookahead does not work or when it is not efficient, these cases will be discussed in another section. Interestingly, if you actually implement this example with k=1, it will match "ab" but not "ac" because ANTLR matches whichever of the ambiguous rules are defined first.

The *filter=true* option sets filtering on, which means that the lexer will ignore all input that does not exactly match one of the non-protected rules (protected rules can only be called by other rules). It can also be set to a protected rule name to assign handling of non-matches to a particular rule, for example:

```
options { filter=BLAH; }
protected BLAH : { _ttype = Token.SKIP; } ;
```

Has the same effect as setting filter to true. However, using the filter as I have just shown is *not* recommended as this kind of use would be entirely redundant.

```
ALPHA   : ('a'..'z'|'A'..'Z')+
        { System.out.println("Found alpha: "+getText()); }
        ;
```

This is an example of a lexer rule, it matches any sequence of one or more characters indicated by the ranges specified. The '+' means match one or more of whatever was specified in the group preceding it, in this case either anything within the range of characters *'a'..'z'* or *'A'..'Z'*, the ranges are separated by a '|' character which signifies a logical OR. An action has also been specified for this match which will only be executed upon a successful match, the action is simply a print statement that indicates that an alpha type token has been found and then the token text is printed with a call to *getText()* which returns the encapsulating tokens text data.

```
NUMERIC : ('0'..'9')+
        { System.out.println("Found numeric: "+getText()); }
        ;
```

This is very similar to the *ALPHA* rule but instead matches sequences of one or more characters within the range *'0'..'9'*, if a match occurs it prints a message indicating it has found a numeric token type and then prints the token's text.

```
EXIT    : '.' { System.exit(0); } ;
```

This matches the literal character '.', (when unquoted '.' can be within rules; as a wild-card that will "match any character" encountered. The action performed when it is matched is to exit the program with an exit status of zero signifying that the program exited normally.

Note

When using the *filter=true* option, if one is processing actual files then one should always include a rule to match newlines, because the lexer needs to be told that a newline has occurred in order to increment the line count, otherwise the lexer would be stuck on one line! A typical newline rule, showing the call to the *newline()* method to handle the newlines

correctly is shown below:

```
NEWLINE   :  ( "\r\n" // DOS
               | '\r'   // MAC
               | '\n'   // Unix
             )
             { newline();
               $setType(Token.SKIP);
             }
          ;
```

The ANTLR directive *$setType(tokenType)* is used to indicate that these sequences of characters should be ignored.

After defining our lexer, ANTLR s ran on the source file to generate the various Java (or C++ or Sather) files:

`java antlr.Tool simple.g`

The command displays the version information and then, if no errors are present in the source file, generates the output files. If errors are present in the source file and are detected then nice messages are displayed describing them. If the errors are not fatal, the files will still be produced and usually some kind of output will be produced, it's just that the output files may not compile if errors were encountered. The text produced when error are encountered can be quite informative such as this one, generated when I deliberately omitted, the rule terminating ';' character, from the end of a rule:

```
ANTLR Parser Generator   Version 2.7.1   1989-2000 jGuru.com
simple.g:13: warning:did you forget to terminate previous rule?
```

The line number indicated is the start of the rule that follows the one I forgot to terminate. In this case, ANTLR recovered from this error and the output files still worked as desired. The output files produced are `SimpleLexer.java` and `SimpleLexerTokenTypes.java`.

`SimpleLexer.java`, the lexer produced implements TokenStream which means that it will return the next token in the token stream when somebody calls *nextToken()* from an instantiation of SimpleLexer (or whatever the lexer's name is). It contains methods which are discussed in more detail in the ANTLR documentation.

`SimpleLexerTokenTypes.java` contains the token type definitions defined as integer constants, for efficient comparison at runtime. Token tables are also used for type checking before translation. In order to use the lexer some kind of program must invoke it, here is a Java program which does just that:

```
import java.io.*;
public class Main {
   public static void main(String[] args) {
      SimpleLexer simpleLexer = new SimpleLexer(System.in);
      while(true) {
         try {
           simpleLexer.nextToken();
         } catch(Exception e) {}
      }
   }
}
```

A new instance of *SimpleLexer* is created with the constructor utilising the *SimpleLexer(InputStream in)* constructor that is automatically generated. An infinite loop is then entered which keeps grabbing the next token from the input stream. The input stream is *System.in* which means that an input interface will be presented on the command line and input will be passed to the lexer every time the enter key is hit (hence no need for newline handling).

All of the java files are compiled by issuing:

`javac *.java`

The `Main.class` produced is executed by issuing:

`java Main`

An example session using this Lexer is shown below:

```
This Lexer recognises strings and numbers: hello 22 goodbye 33
Found alpha: This
Found alpha: Lexer
Found alpha: recognises
Found alpha: strings
Found alpha: and
Found alpha: numbers
Found alpha: hello
Found numeric: 22
Found alpha: goodbye
Found numeric: 33
It ignores everything else: -=+/#
Found alpha: It
Found alpha: ignores
```

```
Found alpha: everything
Found alpha: else
.
```

This Lexer exclusively recognises alpha and numeric content, if it is passed the string "11aa33hi" it will not treat it as a single string but will break up the alpha and numeric parts, as it was specified to do:

```
11aa33hi
Found numeric: 11
Found alpha: aa
Found numeric: 33
Found alpha: hi
.
```

That about wraps up this Lexer introduction but it should be noted that usually a Lexer is used in combination with a Parser, this is example is totally contrived to illustrate some of the concepts. You should consult the ANTLR documentation and check out the other examples in this text for a more thorough comprehension of the Lexers part in the translation process.

7. Simple Lexer/Parser Example

A simple lexer and parser will be discussed. The Job of the lexer will be to tokenise the input stream into the tokens; *NAME*, *AGE*, *DOB*(Date Of Birth) and *SEMI*(semicolon). The parsers job will be to recognise the sequence *DOB NAME AGE(SEMI)* and output this as *Name: name*, *Age: nn*, *DOB nn/nn/nn*. The lexer and parser will be defined in one file simple2.g [files/simple2.g]:

```
class SimpleParser extends Parser;

entry : (d:DOB n:NAME a:AGE(SEMI)
      {
        System.out.println(
          "Name: "    +
          n.getText() +
          ", Age: "   +
          a.getText() +
          ", DOB: "   +
          d.getText()
        );
      })*
      ;

class SimpleLexer extends Lexer;

NAME : ('a'..'z'|'A'..'Z')+;

DOB  : ('0'..'9' '0'..'9' '/')=>
       (('0'..'9')('0'..'9')'/')(('0'..'9')('0'..'9')'/')('0'..'9')('0'..'9')
     | ('0'..'9')+      { $setType(AGE); } ;

WS      :
    (' '
    | '\t'
    | '\r' '\n' { newline(); }
    | '\n'      { newline(); }
    )
    { $setType(Token.SKIP); }
  ;

SEMI : ';' ;
```

The parser is the first class to be specified, however, order does not matter. It is considered better practice by some to start at the most abstract level possible and then work toward the bottom, i.e top down.

```
entry : (d:DOB n:NAME a:AGE(SEMI)
      {
        System.out.println(
          "Name: "    +
          n.getText() +
          ", Age: "   +
          a.getText() +
          ", DOB: "   +
          d.getText()
        );
      })*
      ;
```

You can see that the first rule is *entry*, this is the rule which will be called from the *main* method to start the parsing the input. It says that the parser should look for a *DOB* token followed by a space followed by a *NAME* token, followed by a space followed by an *AGE* token and terminated with a *SEMI* token which is a semicolon (the reason for *SEMI* being in brackets is so that it can be

placed immediately after *AGE* without ANTLR thinking we are trying to reference some token called *AGESEMI*, it is looking for:

```
DOB NAME AGE;
```

If it is successful in finding this sequence of tokens, the variables indicated immediately before the token names (*a* in *a:AGE* etc), will take on the values of the tokens they prefix then an action will be performed, this is indicated by the opening brace, the intended action is to print "Name: name, Age: nn, DOB: nn/nn/nn" where "n" signifies some digit. Notice the scope of the opening parentheses, it's matching parentheses occurs immediately after the closing brace of the action section. It is postfixed with a '*' meaning that this sequence should be matched zero or more times. Let's take a look at the definitions of these tokens in the Lexer:

```
NAME : ('a'..'z'|'A'..'Z')+;
```

A simple sequence of one or more lower-case *OR* upper-case letters.

```
DOB  : ('0'..'9' '0'..'9' '/')=>
       (('0'..'9')('0'..'9')'/')(('0'..'9')('0'..'9')'/')('0'..'9')('0'..'9')
     | ('0'..'9')+      { $setType(AGE); } ;
```

If *DOB* and *AGE* had been specified like this:

```
DOB : ('0'..'9')('0'..'9')'/' ('0'..'9')('0'..'9')'/' ('0'..'9')('0'..'9') ;
AGE : ('0'..'9')+ ;
```

ANTLR would have issued the warning:

```
warning: lexical nondeterminism between rules DOB and AGE upon
simple2.g:0:  k==1:'0'..'9'
```

This is because both of the rules mentioned begin with ('0'..'9'), this means (with a lookahead of one), if the lexer encountered a digit, it would not know which of the statements to try so It would have to try the first one and then *AGE* would never be checked and the parser could never find the sequence it was looking for.

One way around this would be to specify a parser lookahead of 3, that is, include an options statement for the parser like:

```
options { k=3; }
```

This would enable the parser to look forward a maximum of three where it is necessary to disambiguate and it would see that if it encountered two digits followed by a forward slash then it should predict the *DOB* token route and if it does not match that third lookahead value with a forward slash to choose the *AGE* route.

If there are a lot of alternatives all that can be distinguished by changing the lookahead then it is preferable to use this method of increasing the lookahead value but if there is only one or very few alternatives then it is preferable to use a syntactic predicate instead which is what is used in the example. The syntactic predicate in the example is:

```
DOB  : ('0'..'9' '0'..'9' '/')=>
```

This says that the lexer should see if it can find two digits followed by a a forward slash, if it can then the lexer should go on to try to match the sequence specified:

```
(('0'..'9')('0'..'9')'/')(('0'..'9')('0'..'9')'/')('0'..'9')('0'..'9')
```

If it is successful in doing so then the token will be matched as *DOB*, the original rule. If the syntactic predicate fails and it does not match two digits followed by a forward slash then the lexer should try to match the rule specified as the first alternative, after the '|':

```
| ('0'..'9')+      { $setType(AGE); } ;
```

The implication is that if the rule is matched then the action specified after will be executed, which is a call to the *$setType(type)* ANTLR directive, note that this is not a Java statement but an ANTLR one and is prefixed with a '$'. It has been shown how a syntactic predicate can disambiguate between two rules that begin with the same characters.

```
WS      :
    (' '
    | '\t'
    | '\r' '\n' { newline(); }
```

```
    | '\n'      { newline(); }
    )
    { $setType(Token.SKIP); }
  ;
```

The *WS* rule matches white space, hence "WS". The rule will match a space (' ') *OR* a tab ('\t') *OR* a carriage return followed by newline (DOS NEWLINE) ('\r' '\n') *OR* a Unix newline ('\n') delimited via a single newline character, notice that the DOS and Unix newline alternatives have actions associated with them, these actions are calls to the *newline()* method which tells ANTLR to bump up it's line count and goto the next line. Without this the lexer would be stuck on one line! All the alternatives are grouped together within parenthesis and they have an overall action associated with them, this is to set the token type to the ANTLR special type *Token.SKIP* which causes the tokens to be ignored.

```
SEMI : ';' ;
```

This simply matches a semicolon (';').

The Lexer and Parser are setup using a main method, this could have been included as a block of code within the Parser block, (or lexer block) in the form of a *static main* method so that there would be no need to bother writing an extra class. For the sake of clarity, an extra class will be used here:

```
import java.io.*;
public class Main {
  public static void main(String args[]) {
    DataInputStream input = new DataInputStream(System.in);
    SimpleLexer lexer = new SimpleLexer(input);
    SimpleParser parser = new SimpleParser(lexer);
    try {
      parser.entry();
    } catch(Exception e) {}
  }
}
```

Pretty straightforward, instantiation of an input-stream which is passed to the constructor of the lexer and then the lexer is passed to the constructor of the parser. Once the parser has been created, the *entry* method defined in our parser that does all the work is invoked. This is oriented toward receiving input from a file via redirection but one can just as happily input via the prompt that is produced if the main method is invoked without redirecting some input to it. Here is the input file, `test.txt`

```
06/06/82 Peter 20;
03/04/83 Rosie 19;
04/05/81 Mikey 21;
```

After creating the classes with:

`java antlr.Tool Simple2.g`

And compiling everything:

`java *.java`

Main is invoked:

`java Main < test.txt`

This produces the output:

```
Name: Peter, Age: 20, DOB: 06/06/82
Name: Rosie, Age: 19, DOB: 03/04/83
Name: Mikey, Age: 21, DOB: 04/05/81
```

An error can be simulated by changing Mikey's age to "a21", which produces the output:

```
Name: Peter, Age: 20, DOB: 06/06/82
Name: Rosie, Age: 19, DOB: 03/04/83
line 3: expecting AGE, found 'a'
```

8. Expression Evaluation Example

It has come to that time where it is necessary to step into the obligatory expression evaluator example. The expression evaluator will start off simple, more advanced features will then be added. The initial ANTLR grammar for the expression evaluator, expression.g [files/expression.g] is shown below:

```
class ExpressionParser extends Parser;
options { buildAST=true; }
```

```
expr     : sumExpr SEMI!;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : atom (POW^ atom)? ;
atom     : INT ;

class ExpressionLexer extends Lexer;

PLUS  : '+' ;
MINUS : '-' ;
MUL   : '*' ;
DIV   : '/' ;
MOD   : '%' ;
POW   : '^' ;
SEMI  : ';' ;
protected DIGIT : '0'..'9' ;
INT   : (DIGIT)+ ;

{import java.lang.Math;}
class ExpressionTreeWalker extends TreeParser;

expr returns [double r]
  { double a,b; r=0; }

  : #(PLUS  a=expr b=expr)  { r=a+b; }
  | #(MINUS a=expr b=expr)  { r=a-b; }
  | #(MUL   a=expr b=expr)  { r=a*b; }
  | #(DIV   a=expr b=expr)  { r=a/b; }
  | #(MOD   a=expr b=expr)  { r=a%b; }
  | #(POW   a=expr b=expr)  { r=Math.pow(a,b); }
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
  ;
```

The grammar definition begins with the parser section, this will be explained step by step.

```
class ExpressionParser extends Parser;
options { buildAST=true; }
```

The class is declared normally and then the option *buildAST=true* is specified, this signifies that the parser should build an AST(Abstract Syntax Tree) as it parses the input tokens. Special notation will be used to specify how the tree should be build up.

```
expr     : sumExpr SEMI!;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
```

The top level rule of the expression is *expr*, this simply references the next element down, indicating that an *expr* can be a *sumExpr*, the rule also specifies that an expression is terminated with a *SEMI*. This rule is redundant, in that, the top rule could have been *sumExpr* since *expr* references it directly, the addition being *SEMI!* (which would have to be appended to *prodExpr*. The reason for this is that it makes it clearer that the overall thing being matched is an expression and not a sum expression. The terminating *SEMI* is postfixed with a '!', this tells the AST builder not to include this token in the tree (whereupon it would postfix the whole expression).

The *sumExpr* is an expression that consists of a *prodExpr* followed by zero or more (*PLUS* OR *MINUS*) *prodExpr*, sequences, this is so that *sumExpr* can recognise sequences such as:

```
prodExprPLUSprodExprMINUSprodExpr
```

Because zero or more of these sequences must be present for a match, expressions without *PLUS* and *MINUS* can be constructed because an expression can just be a *prodExpr* without the additions. This will form a hierarchy whereby due to this kind of optionality (zero or more), an expression can just be an *atom* which is an *INT*, this will be explained in more detail later.

An AST is a special kind of tree that can have an arbitrary number of subtrees (children) which are ASTs themselves. When walking the tree one can manipulate the order in which nodes are visited with all the expressiveness of the implementation language (Java, C++ or Sather).

The *PLUS* and *MINUS* token references are postfixed with the caret ('^') character. This is an ANTLR directive specific to the creation of AST's, it specifies that the token the caret postfixes should become the root of the current AST or AST subtree. The first caret-postfixed token encountered in an expression being evaluated will become the root of the AST overall. If a caret-postfixed token is encountered whilst evaluating a child of a root, the child element will become a new subtree with a root equal to the caret-postfixed token.

The AST structure is defined in such a way that operator precedence is obeyed. In this case the tree will be constructed such that the a root will represent an operator and the children will represent it's operands. When the tree is parsed it will be parsed in an ordinary fashion, evaluating it's children from left to right recursively. The rule:

```
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
```

Dictates that the first and second children of a *sumExpr* must be *prodExpr*'s. It is because of this that the desired precedence is guaranteed. Take a look at the *prodExpr* rule:

```
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
```

This says that a *prodExpr* must consist of a *powExpr* followed by zero or more (*MUL* OR *DIV* OR *MOD* AND *powExpr*) sequences. Multiplication, division and modulo have been grouped together because they have equal precedence. The caret ('^') is used again to specify that the root of this subtree will be the operator that is specified in the expression, hence the first and second children can only be *powExpr*'s.

The children of the root will be evaluated before applying them to the root so the *powExpr*'s will be evaluated first, hence the value of the *powExpr* will be calculated before applying the operator of the *prodExpr* which is the desired course of action since a power expression has a higher precedence than a product expression, which in turn has a higher precedence than a sum expression hence the evaluation order of the expression is determined by the structure of the AST created which in turn forces the desired precedences.

```
powExpr  : atom (POW^ atom)? ;
atom     : INT ;
```

It can be seen that a *powExpr* consists of an atom followed by an optional (*POW atom*) sequence. This means that the sequence must occur zero or one times. Why is this rule not defined as zero or more times like this:

```
powExpr  : atom (POW^ atom)* ;
```

Because this *powExpr* is broken. What happens if more than one power is specified? The order of evaluation will occur from left to right but it should occur from right to left (amongst the *POW*s) because an expression such as *3^2^2* should evaluate as *(3^(2^2))* but this will be evaluated as *((3^2)^2)*, a fix for this will be examined later. By only allowing a maximum of one (*POW atom*) sequence, this problem is not encountered but the user is limited to a single level of exponential. Since a multiple exponential is the same as a single exponential equal to the product of the individual exponentials, this should not be a problem.

The point is that the subtrees will be evaluated first, so the AST is created so that AST subtrees that define operators of higher precedence are only allowed to be children of AST subtrees that define operators of a lower precedence so that the operation of highest precedence is always evaluated first. It can be seen that the final rule is for an *atom* which is an *INT*. So the simplest possible expression would be a single integer.

Note

You may find it easier to think of the AST defined here as being like a binary tree because each of the operators only has two children, this is more intuitive and perhaps easier to understand. Effectively, maintaining the analogy, the left and right subtrees of the root will be evaluated before applying them to the root of the tree, that is the tree would be ns-traversed in a postfix manner; left, right, root.

Let's take another look at the parser in one block:

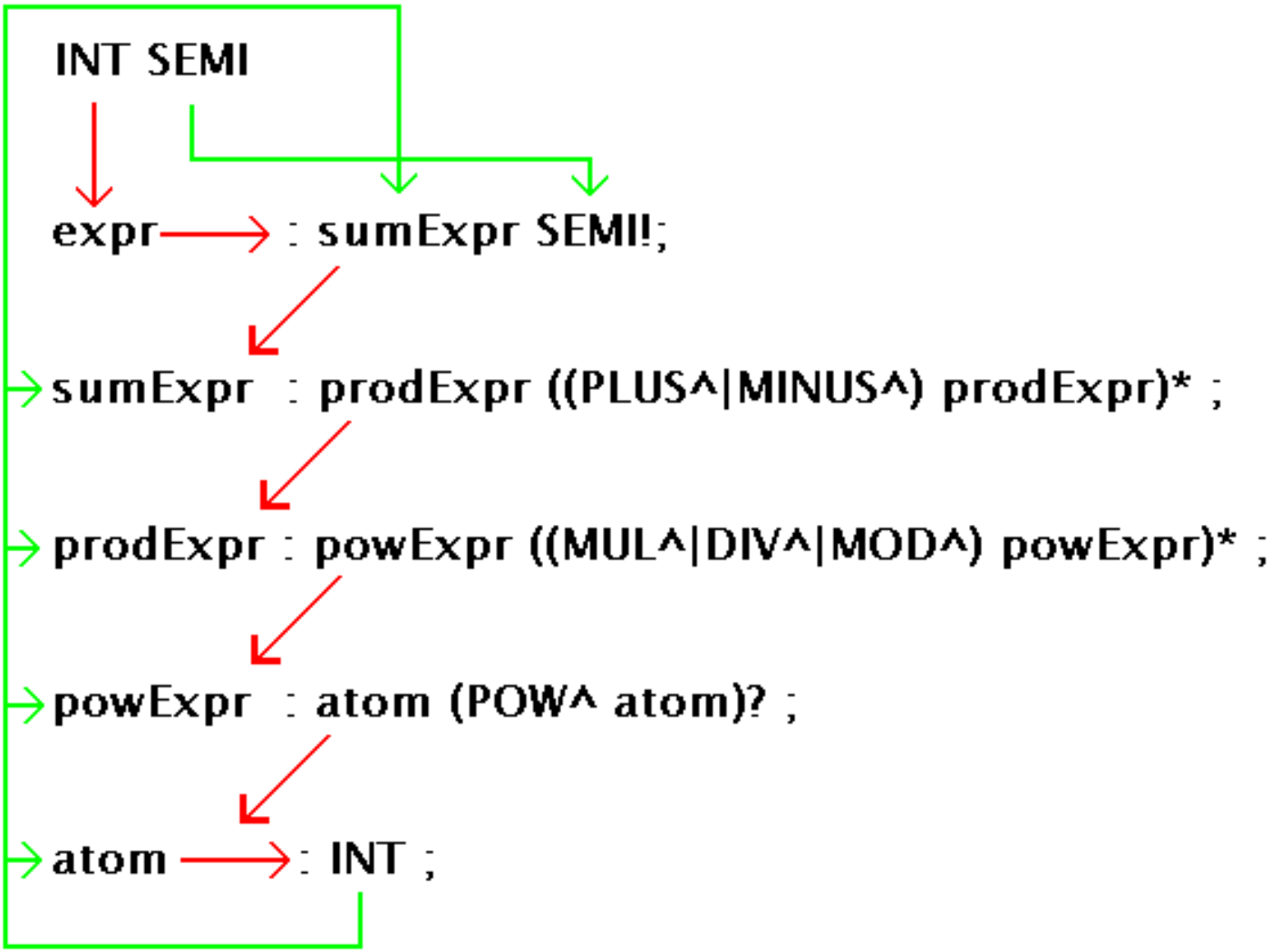
```
class ExpressionParser extends Parser;
options { buildAST=true; }

expr     : sumExpr SEMI!;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : atom (POW^ atom)? ;
atom     : INT ;
```

How can an expression be just an atom? Assume we are trying to match the expression *"5;"*. This consists of the tokens *INT* and *SEMI*. The parser will try to match the token *INT* with one of it's rules. It looks at the first rule, *expr*, and sees the subrule, *sumExpr*, referenced so it takes a look at that. The first component of *sumExpr* is *prodExpr* and so the parser tries to match the token against *prodExpr*. The first component of *prodExpr* is *powExpr* so the parser tries to match the token against *powExpr*.

The first component of *powExpr* is an *atom*so the parser tries to match the token against *atom*, *atom* is defined as consisting of the token *INT* which is what the parser is looking for so *atom* is matched which means that *powExpr* is matched, which means that *prodExpr* is matched which means that *sumExpr* is matched which means that the first component of *expr* has been matched. The next token to be matched is *SEMI* which matches the second component of *expr* so *expr* is matched. Here is a diagram which attempts to illustrate this:

Figure 1. *INT SEMI* Match

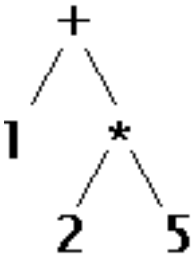


The red lines (or dark lines if you are in greyscale), are supposed to illustrate failed matches and the line directions show how the order the parser checks the rules. The green lines are supposed to illustrate correct matches. The line from *SEMI* to *SEMI* however is an exception, and is green because I wanted to illustrate that *SEMI* was a match, I suppose that the first line from *INT* is not really a fail either but if another colour had been used the diagram would have looked odd.

Do the rules handle precedence correctly? Let's take a look. Imagine that the expression *1+2*5;*, we know that this should evaluate as *(1+(2*5)* and not *((1+2)*5)*. This is the sequence *((INT)(PLUS)(INT)(MUL)(INT)(SEMI)*.

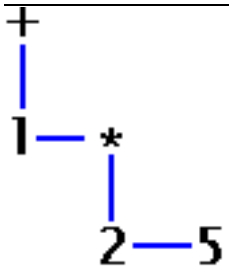
The parser can only generate the tree:

Figure 2. 1+(2*5) Binary Tree



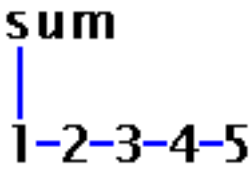
Because a *sumExpr* cannot be a subtree of a *prodExpr*, as is defined in the parser grammar. It is because of the defined structure that precedence rules are maintained. Here is the AST version of the tree shown above:

Figure 3. 1+(2*5) AST



An AST can have an arbitrary number of children, the children are referred to as "siblings" with respect to each other. One can develop trees such as:

Figure 4. Sum AST



Where *sum* is some function that accepts multiple arguments and returns the sum of the arguments, you would of course specify what to do when encountering a particular node of the tree by using a treeparser of some sort, this example has a treeparser (called a treewalker because the tree is walked) which is discussed later. The next section of the grammar is the lexer:

```
class ExpressionLexer extends Lexer;

PLUS  : '+' ;
MINUS : '-' ;
MUL   : '*' ;
DIV   : '/' ;
MOD   : '%' ;
POW   : '^' ;
SEMI  : ';' ;
protected DIGIT : '0'..'9' ;
INT   : (DIGIT)+ ;
```

These rules are pretty self-explanatory and define the tokens used in the parser just described. *DIGIT* is a protected rule, this means it cannot be referenced externally, only by internal rules. It is used in the definition of the *INT* rule which says, "one or more digits". If protected was not specified ANTLR would generate a nondeterminism error between *DIGIT* and *INT*.

The next section of the grammar is the tree parser:

```
{import java.lang.Math;}
class ExpressionTreeWalker extends TreeParser;

expr returns [double r]
  { double a,b; r=0; }

  : #(PLUS a=expr b=expr)  { r=a+b; }
  | #(MINUS a=expr b=expr) { r=a-b; }
  | #(MUL  a=expr b=expr)  { r=a*b; }
  | #(DIV  a=expr b=expr)  { r=a/b; }
  | #(MOD  a=expr b=expr)  { r=a%b; }
  | #(POW  a=expr b=expr)  { r=Math.pow(a,b); }
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
  ;
```

The line before the the class declaration is a header that will be pulled into the generated Java file immediately before the class declaration in the generated file. Whitespace is significant between the opening and closing braces hence no gaps have been left because putting excess spaces there is unnecessary. *java.Math* is imported so the Java Math classes can be used. Following the class declaration is the *expr* rule definition that returns the *double*, *r*:

```
expr returns [double r]
  { double a,b; r=0; }
```

Immediately preceding the opening of the rule is a Java code section which defines two doubles, a and b, and intialises r to zero.

```
  : #(PLUS  a=expr b=expr)  { r=a+b; }
  | #(MINUS a=expr b=expr)  { r=a-b; }
  | #(MUL   a=expr b=expr)  { r=a*b; }
```

```
  | #(DIV   a=expr b=expr)  { r=a/b; }
  | #(MOD   a=expr b=expr)  { r=a%b; }
  | #(POW   a=expr b=expr)  { r=Math.pow(a,b); }
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
  ;
```

The rule definitions all take use the AST syntax, that is:

```
      #(ROOT child.1 child.2 ... child.n);
```

The first rule says, if *PLUS* is found as a root, assign the values of the evaluation of the two child subtrees to the variables '*a*' and *b* respectively. The rule does not literally say "evaluate the subtrees", this will happen automatically due to the fact that the rule says to match a tree with *PLUS* as a root that has two *expr*'s as children. It is in the matching of these children, in order to match the whole rule, that the subtrees will be evaluated. The action specified upon a successful match, is to set *r* equal to *a+b*.

In this case, it is obvious with a lookahead of one which rule to match if *PLUS* is found as a root because this is the only rule that has *PLUS* as the first element. This may not always be the case, consider adding the ability to specify the sign of a number (as many times as you like, -+-+-5), then the *PLUS* and *MINUS* tokens would not be used exclusively for the dyadic addition rule but would also occur as the first element of the monadic sign rule. With a lookahead of one there would be conflicts, this issue is discussed later.

```
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
```

This alternative is more interesting than the others so will get it's own special mentioning. It simply assigns to *r*, the value of the *INT* found. This handles the "base case", as such.

The treeparser, is being used to walk the tree and evaluate the expressions entered. Let's look at how this all fits together, here is Main.java [files/Main.java]:

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
public class Main {
  public static void main(String args[]) {
    try {
      DataInputStream input = new DataInputStream(System.in);

      ExpressionLexer lexer = new ExpressionLexer(input);

      ExpressionParser parser = new ExpressionParser(lexer);
      parser.expr();

      CommonAST parseTree = (CommonAST)parser.getAST();
      System.out.println(parseTree.toStringList());
      ASTFrame frame = new ASTFrame("The tree", parseTree);
      frame.setVisible(true);

      ExpressionTreeWalker walker = new ExpressionTreeWalker();
      double r = walker.expr(parseTree);
      System.out.println("Value: "+r);
    } catch(Exception e) { System.err.println("Exception: "+e); }
  }
}
```

First we import all the necessary classes and open the class and *static main* method as usual, the contents of *main* are wrapped within a *try...catch* statement to catch any errors generated, if there are any errors, the error is printed to stdout.

```
      DataInputStream input = new DataInputStream(System.in);
```

A *DataInputStream* is setup.

```
      ExpressionLexer lexer = new ExpressionLexer(input);
```

The lexer is created and told to accept data from the inputstream.

```
      ExpressionParser parser = new ExpressionParser(lexer);
      parser.expr();
```

The parser is created, using the lexer to deliver tokens. The *expr()* method is called which tells the parser to match an expression as defined by the parser rules.

```
CommonAST parseTree = (CommonAST)parser.getAST();
```

Remember that *options { buildAST=true; }* was specified? Here a *CommonAST* object is created and assigned a reference to the AST created by the parser's *expr* rule. It has to be downcast from *collections.AST*.

```
System.out.println(parseTree.toStringList());
```

The AST is printed using the *CommonAST toStringList()* method.

```
ASTFrame frame = new ASTFrame("The tree", parseTree);
frame.setVisible(true);
```

A new ASTFrame is created, which is a frame designed for viewing AST's imported from *antlr.debug.misc.ASTFrame*. The frame is created with the title "The tree" and the *CommonAST* object created before. The frame is made visible, this will generate a frame showing the AST.

```
double r = walker.expr(parseTree);
System.out.println("Value: "+r);
```

A new *double* is defined and assigned the value of the expression provided by calling the *expr* rule that was defined for the TreeParser created (which is translated to a Java method). The AST is passed as an argument, finally the value of the expression is printed to stdout.

Let's go through an example expression to see what output is generated. Assume that all classes have been generated by running ANTLR on the grammar. The expression:

```
1+2-3*4/5^6;
```

Is placed in a file called `test.txt` and used as input to Main:

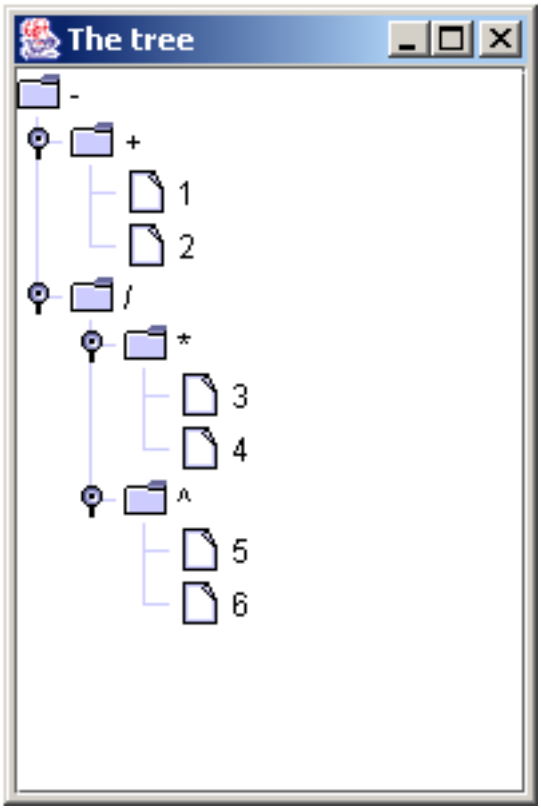
```
java Main < test.txt
```

The tree expressed as a list via *toStringList()* is output:

```
( - ( + 1 2 ) ( / ( * 3 4 ) ( ^ 5 6 ) ) ) ;
```

The AST comes up in the ASTFrame:

Figure 5. AST of (1+2-3*4/5^6)



And the value is output:

```
Value: 2.999232
```

That concludes the basic expression evaluator, the next section will discuss some extensions.

9. Extending The Expression Evaluator

9.1. Nested Expressions

The old expression evaluator did not allow nesting of brackets, in fact brackets were not mentioned at all. Try to think how you would add nested brackets to the evaluator. How would you get the evaluator to create an AST such that the innermost nested brackets are evaluated first? The solution is quite simple, possibly a little subtle? First of all lets take a look at the content of the old parser:

```
expr     : sumExpr SEMI;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)*;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : atom (POW^ atom)? ;
atom     : INT ;
```

powExpr has a higher precedence than *prodExpr* and is made to be a child of a *prodExpr* because of the tree structure. During a transversal of the tree, the children will be evaluated independently of the root because the values of the evaluation of the children are assigned to the variables *a* and *b* before the operation is applied and the result returned:

```
#(PLUS  a=expr b=expr)  { r=a+b; }
```

See that *a* and *b* have to be evaluated first because the operation is only executed after a successful match of the rule which means that both *a* and *b* must have been evaluated. The most basic component of this AST is an *atom*. We want to say that an atom can also be another expression, one can not just redefine atom to equal:

```
atom     : INT | expr ;
```

This would generate some infinite recursion errors:

```
expression.g:8: infinite recursion to rule sumExpr from rule atom
expression.g:7: infinite recursion to rule sumExpr from rule powExpr
expression.g:6: infinite recursion to rule sumExpr from rule prodExpr
expression.g:5: infinite recursion to rule sumExpr from rule sumExpr
expression.g:8: infinite recursion to rule sumExpr from rule atom
```

Imagine the case of an out of place token. The parser would check the token against all the rules and find that it did not make a match, it would reach the bottom and check it against *atom*, it would find that *INT* did not match so it would then check it against *expr* which would cause another check through all the rules and so on, forever. So *expr* has to be redefined as:

```
expr     : LPAREN^ sumExpr RPAREN! ;
```

This redefinition is crucial, *LPAREN* and *RPAREN* are tokens defined in the lexer and represent *'('* and *')'* respectively. This rule says to match *LPAREN* followed by a *sumExpr* followed by *RPAREN*. *LPAREN* is postfixed with a caret ('^') which indicates that in the generated AST *LPAREN* should become a tree root and have the child *sumExpr*, *RPAREN* does not become a child because it is prefixed with an exclamation mark, this is because it is unnecessary in the final AST. *SEMI* was also removed so that nested expressions do not have to be terminated by a *SEMI*.

If the out of place token arose again, the parser could not get in an infinite loop because the *expr* rule begins with a *LPAREN*, hence if the token did not match *LPAREN* the parser would immediately throw a noViableAltException. There is no way the rogue token could get past the *expr* rule. Here is the new parser definition in one block:

```
expr     : LPAREN^ sumExpr RPAREN! ;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : atom (POW^ atom)? ;
atom     : INT | expr ;
```

So now an *expr* can recursively contain as many *expr*'s as is desired. What should the treeparser do when it encounters a tree or subtree with *LPAREN* as root and an *expr* as a child? The desired action is to return the value of the evaluation of the child. The addition of this rule into the tree parser achieves this:

```
  | #(LPAREN a=expr) { r=a;}
```

This rule matches an AST with a *LPAREN* as root and an *expr* as a child. The result of the evaluation of *expr* is assigned to the *a* variable which in turn is assigned to the *r* variable, hence *r* will not receive a value until *a* does and *a* will not receive a value until *expr* has been matched. Causing a knock on evaluation of all subtrees of *expr* in order to match the original *expr*, this may include evaluation of other *expr*"s.

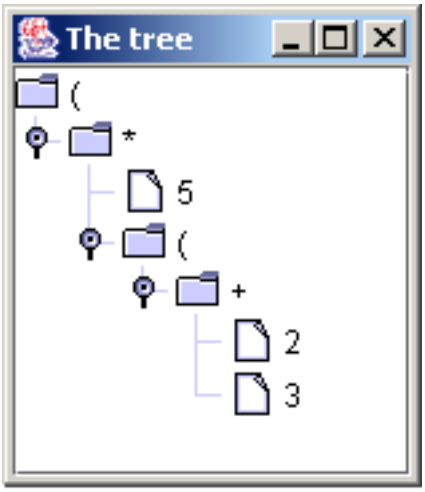
Eventually all leaves of the AST must be *INT*'s, unless an infinite number of sub expressions were contained within the master expression, this is unlikely. It is assumed that the person creating the expression has their sanity intact and is not attempting to generate some kind of crazy AST that will keep calling *expr* billions of recursive levels down, in order to match some infinite order of nested *expr*', pertaining to the match of a master *expr*, I am not sure if this is even possible.

In the normal world, the calls to *expr* will return a final value as a result of all these *INT* leaves being used in the various operations specified in the expression. The values of these operations will work there way up the recursive levels until eventually the original *expr* has been matched, whereupon, *a* will be assigned the value of the *expr*, *r* will be assigned the value of *a* and *r* will be returned.

To restate: Whenever an *LPAREN* is encountered, the sub-expression will be evaluated, the evaluation of the sub-expression will return a value which will then be used in the evaluation of the *expr* that had the sub-expression as a child. The result evaluation of this, may in turn, be used in the evaluation of the parent *expr* until no more parent *expr*'s are present and the master *expr* has been evaluated.

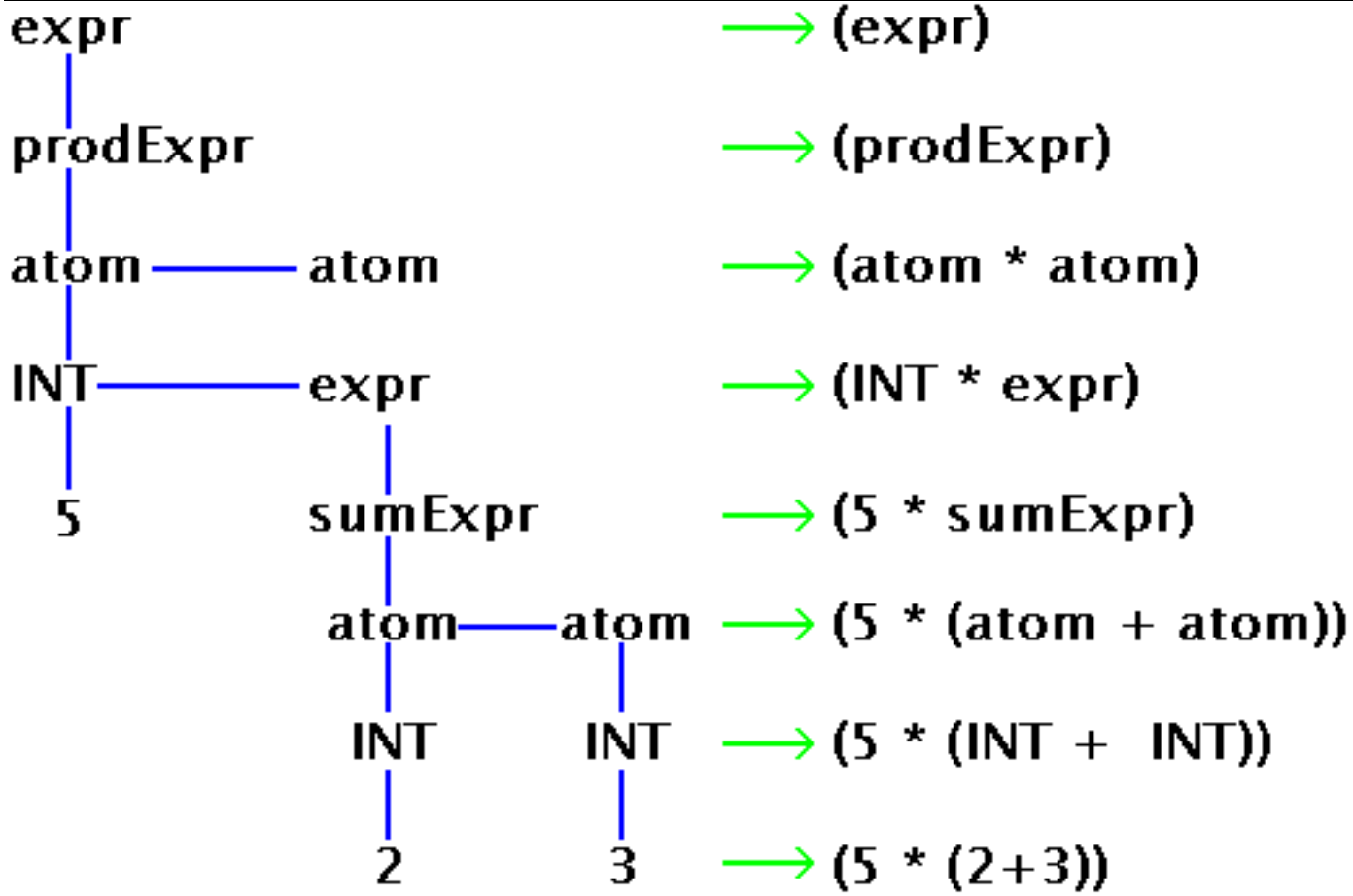
An AST illustrating this should help to clarify things, here is an AST of the expression *(5*(2+3))*:

Figure 6. AST of (5*(2+3))



The root of the tree is an *expr* signified by the open bracket '*LPAREN*' as expected. It's child is a *prodExpr* whose first child is an *atom* which is an *INT* equal to five and whose second child is another *atom* but this time a *expr* whose child is a *prodExpr* containing two children which are both *atom*'s and are *INT*'s which are equals to the values two and three respectively. Here is another illustration of this AST to help clarify things:

Figure 7. Another View of (5*(2+3))



Here is the whole grammar: expression2.g [files/expression2.g]. The *Main* method used to run it is the same as for the original expression evaluator.

Note

Adding subexpressions also solves the problem of being limited to a single level of exponential, the ambiguous expression *(2^5^5)* can be respecified as *(2^(5^))* to get the desired result.

9.2. Adding The Sign Operator

The sign operator has a higher precedence than any of the operators already defined in this expression evaluator, hence, it's rule will occur just above *atom*. There is no need to use syntactic predicates to distinguish between dyadic and monadic use of the *PLUS* and *MINUS* operators, this is implied by the context in which the expression is used. If trying to match the dyadic use of the *PLUS* and *MINUS* operators, the parser will expect to see the operator infixed between two *prodExpr*'s, where as, in the monadic sense, the parser will expect to see the operator prefixing an *atom*. Here is how it is done:

```
imaginaryTokenDefinitions :
   SIGN_MINUS
   SIGN_PLUS
;

expr     : LPAREN^ sumExpr RPAREN! ;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^|MOD^) powExpr)* ;
powExpr  : signExpr (POW^ signExpr)? ;
signExpr : (
         m:MINUS^ {#m.setType(SIGN_MINUS);}
         | p:PLUS^  {#p.setType(SIGN_PLUS);}
         )? atom ;
atom     : INT | expr ;
```

Ignore the imaginary token business for now, this will be explained later.

Consider the matching of the expression *(-3- -2)*. The first token is *LPAREN* which is matched by *expr*. The second token is *MINUS* so the parser checks whether this token can be a *sumExpr* which is the rule it is trying to match to match the *expr* that has just been opened. Is the first token of *sumExpr* a *MINUS*? No, it's a subrule so *prodExpr* is called to find out if the first token of that rule is a *MINUS*. Is the first token of *prodExpr* a *MINUS*? No, it's a subrule so *powExpr* is called to find out if the first token of that rule is a *MINUS*. Is the first token of *powExpr* a *MINUS*? No, it's a subrule so *signExpr*is called to find out if the first token of that rule is a *MINUS*.

Is the first token of *signExpr* a *MINUS*? Yes, it is! Brilliant, but, the *MINUS* must be followed by an *atom* for *signExpr* to match. The next token is an *INT*(3) which *is* an *atom* so finally something has been matched, a *signExpr* has been matched. Hang-on, let's not get carried away here, this whole process stemmed from the fact that the parser was trying to match the first token of *sumExpr* because it's parent *expr* must contain one. This caused calls to all the subrules ending up at *signExpr*. Now that a match has occurred the parser works it's way back up the recursive levels and discovers that it has matched a *powExpr* because a *powExpr* may consist of just a *signExpr* which in turn means it has matched a *prodExpr* because a *prodExpr* can consist of just a *powExpr*, then it discovers that it has also matched a *sumExpr* because a *sumExpr* can consist of just a *prodExpr*, effectively it has almost matched an *expr* because if the next token is a *RPAREN* the *expr* would be finished. However, the *sumExpr* rule may not be finished so first the parser must check to see if next token matches the next token of the *sumExpr* rule.

```
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
```

The next token of the *sumExpr* rule is *PLUS* OR *MINUS*, the next token in the expression being matched is *MINUS*, hence the *MINUS* in *prodExpr* is matched. There cannot be any ambiguity here because *signExpr* is not an alternative at this point. There is no rule which goes (*prodExpr signExpr*) so *sumExpr* is the only possible route, if *sumExpr* is not matched, *expr* will not be matched (because *expr* would be expecting *RPAREN*) and the expression will fail. The parser will finish checking whether or not this rule matches before doing anything else anyway, so even if there was a (*prodExpr signExpr*) rule (occurring later in the parser), *sumExpr* would have to fail for that to be checked. There is no other rule so the parser does match the *MINUS* in *sumExpr* and the matching of *sumExpr* goes on as expected.

The parser has just matched the *MINUS* in *sumExpr*, it now tries to match *prodExpr* which is the third component of *sumExpr*. *prodExpr* is matched in exactly the same manner that the first *prodExpr* was matched. This means that *sumExpr* has been matched and the parser has matched the second token of *expr*. The parser now expects *RPAREN* from the lexer, which it gets, and *expr* is successfully matched. Here is a ridiculously over the top illustration of this process [files/images/sillyastthing.png].

Hopefully I have not obfuscated the working of this rule. Let's take a closer look at the workings of it:

```
signExpr : (
         m:MINUS^ {#m.setType(SIGN_MINUS);}
         | p:PLUS^  {#p.setType(SIGN_PLUS);}
         )? atom ;
```

First of all, note that the first part of *signExpr* is enclosed within parentheses postfixed with a question mark indicating that a *signExpr* can be either an *atom* or a sign symbol followed by an atom. The rule says that if a *MINUS* or *PLUS* is encountered, the *MINUS* or *PLUS* should become the root of a new subtree with *atom* as the only child. This is exactly the desired behaviour since by making the *atom*a child, we can recognise the root in the tree from the tree parser and perform some action on this child. This is where the imaginary tokens come in:

```
        m:MINUS^
```

Assigns the root node *MINUS* to the variable '*m*'

```
         {#m.setType(SIGN_MINUS);}
```

Replaces this root node in the tree with the token *SIGN_MINUS* instead of *MINUS*. This is done because the tree is already using the root *MINUS* to recognise that it should perform a dyadic subtraction operation:

```
  | #(MINUS a=expr b=expr) { r=a-b; }
```

In order not to have some kind of syntactic predicate to determine which kind of *MINUS* operation to perform, another token is created to represent the monadic *MINUS* operation called *SIGN_MINUS*. The same thing is done for the *PLUS* operator:

```
imaginaryTokenDefinitions :
   SIGN_MINUS
   SIGN_PLUS
;
```

When parsing the AST, the tree parser knows exactly which kind of operation to perform. The full tree parser section is shown below:

```
{import java.lang.Math;}
class ExpressionTreeWalker extends TreeParser;

expr returns [double r]
  { double a,b; r=0; }

  : #(PLUS  a=expr b=expr) { r=a+b; }
  | #(MINUS a=expr b=expr) { r=a-b; }
  | #(MUL   a=expr b=expr) { r=a*b; }
```

```
  | #(DIV   a=expr b=expr) { r=a/b; }
  | #(MOD   a=expr b=expr) { r=a%b; }
  | #(POW   a=expr b=expr) { r=Math.pow(a,b); }
  | #(LPAREN a=expr)       { r=a; }
  | #(SIGN_MINUS a=expr)   { r=-1*a; }
  | #(SIGN_PLUS  a=expr)   { if(a<0)r=0-a; else r=a; }
  | i:INT { r=(double)Integer.parseInt(i.getText()); }
  ;
```

If a *SIGN_MINUS* root is encountered, the desired consequence is to negate the sign of the operand, this is achieved by multiplying the operand by -1. If a *SIGN_PLUS* root is encountered, the desired consequence is to do nothing if the operand is already positive and to make the operand positive if it is negative, this is achieved by having a conditional statement which subtracts the operand from zero if it is negative and does nothing to the operand otherwise. The whole grammar can be found here: expression3.g [files/expression3.g]. The *Main* method used to run it is the same as for the original expression evaluator.

Note

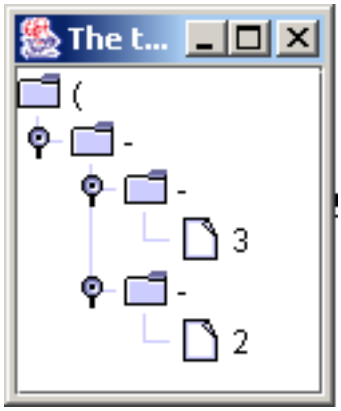
As pointed out to me by Safak Oekmen, this interpretation of *SIGN_PLUS* is quite strange; usually one would not assume that a positive sign prefix would change the sign of a following negative number to positive. However, this doesn't affect the pedagogical implications of the example if *SIGN_PLUS* takes on the indicated role so it will be left as is.

Let's look at an example run through of the expression *(-3- -2)*, it is assumed that ANTLR has been ran on the grammar, all classes compiled and the expression fed to the lexer via *Main*. Here is the stringList and value printed:

```
 ( ( ( - ( - 3 ) ( - 2 ) ) )
Value: -1.0
```

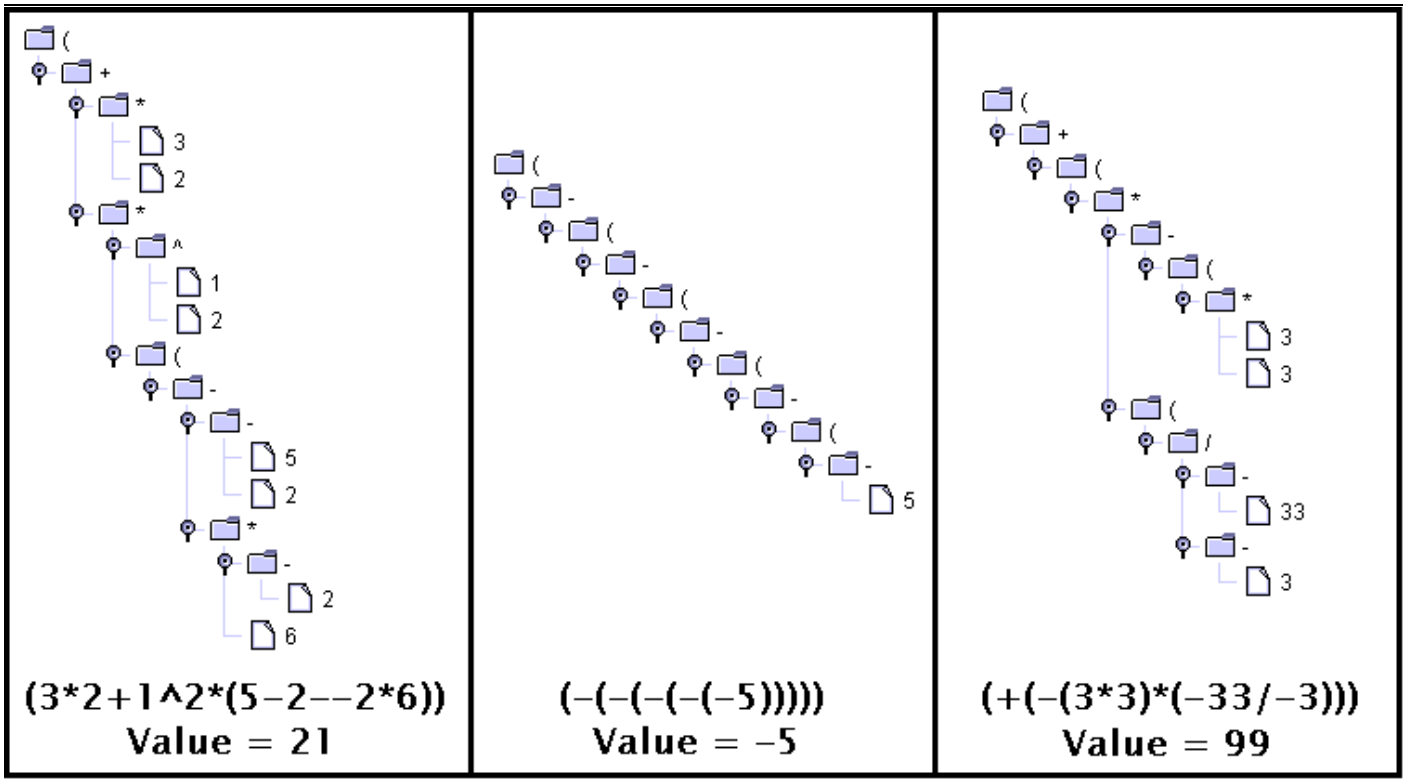
Here is the AST produced:

Figure 8. (-3- -2) AST



Here are some more AST's:

Figure 9. A few AST's



Note

In the parser for the expression evaluator, a top level expression was specified as:

```
expr     : LPAREN^ sumExpr RPAREN! ;
```

If we were reading the expression from a file, this rule should actually be:

```
expr     : LPAREN^ sumExpr RPAREN! EOF! ;
```

So that the EOF(End Of File), token is matched and the whole rule is matched, as it happens, the parser will match as much as it can anyway, so it matches enough to build the AST. The inclusion of EOF in the rule would not allow us to enter expressions at the command line unless one had a way to specify the EOF character. Bear this note in mind when constructing parsers that read from files.

10. A Translation Example - CSV to XHTML Table

A Lexer translates from a stream of characters to a stream of Tokens. A parser recognises certain sequences of Tokens and performs actions based on this recognition such as perhaps converting the sequence to a sequence of machine language instructions to be executed later, or perhaps generating an AST for later transversal. The more usual of the two would be to first generate an AST - another translation - and then this AST would later be parsed itself to generate some other form of output. Compilation can be seen as a series of translations, this section will illustrate the idea of translation with a simple ANTLR example that translates a comma separated variable (CSV) file to an XHTML table.

A CSV is a very simple kind of data structure; variables separated by commas and newlines to create a kind of table. An example will clarify this:

```
"STUDENT ID, "NAME,            "DATE OF BIRTH
"129384,     "Davy  Jones,     "03/04/81
"328649,     "Clare Manstead,  "30/11/81
"237090,     "Richard Stoppit, "22/06/82
"523343,     "Brian Hardwick,  "15/11/81
"908423,     "Sally Brush,     "06/06/81
"328453,     "Elisa Strudel,   "12/09/82
"883632,     "Peter Smith,     "03/05/83
"542033,     "Ryan Alcot,      "04/12/80
```

The translator developed in this section will translate CSV's of this type. The structure of the CSV will be discussed whilst simultaneously developing the parser. This will illustrate the very first translation, that of translating from general ideas about the structure of a file to an ANTLR parser capable of recognising this structure.

The CSV *file* is composed of one or more *line*s and terminates with an EOF token:

```
file   : ( line (NEWLINE line)* (NEWLINE)? EOF )
```

A *line* consists of one or more *record*s, *NEWLINE* is handled by *file*.

```
line   : ( (record)+ ) ;
```

A *record* consists of a *RECORD* token, optionally followed by a *COMMA* token (because tokens at the end of a line or file are not followed by a *COMMA*):

```
record : ( (r:RECORD) (COMMA)? ) ;
```

Notice that the last line of the CSV is an exceptional case because it terminates with *EOF* instead of *NEWLINE*, this is handled by the *file* rule, which says that a file begins with a line, has zero or more (*NEWLINE* line) sequences then has an optional *NEWLINE* and then finally terminates with an *EOF* token.

```
class CSVParser extends Parser;
file   : ( line (NEWLINE line)* (NEWLINE)? EOF )
line   : ( (record)+ ) ;
record : ( (r:RECORD) (COMMA)? ) ;
```

It is coupled with this lexer:

```
class CSVLexer extends Lexer;
options { charVocabulary='\3'..'\377'; }
RECORD  : '"'! (~(','|'\r'|'\n'))+ ;
COMMA   : ',' ;
NEWLINE : ('\r''\n')=> '\r''\n' //DOS
        | '\r'                  //MAC
        | '\n'                  //UNIX
        { newline(); }
        ;
WS      : (' '|'\t') { $setType(Token.SKIP); } ;
```

First of all, *charVocabulary* is set to *'\3'..'\377'*, this defines the set of Unicode characters that characters in the inputstream must belong to. It also implicitly defines which characters will be used as alternatives when an "everything except" kind of rule is specified.

RECORD is an example of an "everything except" rule and says that a *RECORD* consists of one or more characters in the defined character range, except comma, carriage-return or newline. Notice that *RECORD* was defined as starting with a double quote character, this format was chosen so that the user could include records with spaces in with more ease. If this double quote was not used to signify the start of a record, the record would consist of all characters from the start of the record up-to the next comma, this would include spaces. If a nicely aligned CSV like:

```
Dave,    21
Richard, 55
Peter,   98
```

Was used, then the records would be "Dave",

"   21"
, "Richard", "55 ", "Peter" and

"   98"
, which is probably not what is desired. Of course, one could process the tokens afterward to strip leading and trailing spaces, but then what if some of the tokens *should* contain leading or trailing spaces? It was decided that in this example the records would be defined in this way. You could play around with these design issues yourself as a learning experience. One more thing to note about this is that the double quote is postfixed with an exclamation mark so that the double quote itself is not included in the actual text of the token. The rest of the lexer rules are self-explanatory.

If this lexer and parser were used to process a CSV file, nothing would happen except the rules would be matched correctly and the program would terminate with an exit status of zero. For illustration, I added some print statements to the lexer so that it would print out when a rule was called and if it was matched and what the record was:

```
class CSVParser extends Parser;
options { k=2; }
file   {System.out.println("file called");}
       : ( line (NEWLINE line)* (NEWLINE)? EOF)
       {System.out.println("file matched");}
       ;

line   {System.out.println("line called");}
       : ( (record)+ )
       {System.out.println("line matched");}
       ;

record {System.out.println("record called");}
```

```
       : ( (r:RECORD) (COMMA)? )
       {System.out.println("record = " + r.getText());
        System.out.println("record matched");}
       ;
```

Notice the lookahead of two to distinguish between (*NEWLINE* line) and (*NEWLINE*) *EOF*. The parser was ran (via a Main method explained later), on the following file:

```
"David Sprocket, "89
"Cindy Brocket,  "18
"Michael Rocket, "33
```

The output that was produced is shown below (reformatted by hand to make it clearer):

```
file called
  line called
    record called
      record = David Sprocket
    record matched

    record called
      record = 89
    record matched
  line matched

  line called
    record called
      record = Cindy Brocket
    record matched

    record called
      record = 18
    record matched
  line matched

  line called
    record called
      record = Michael Rocket
    record matched

    record called
      record = 33
    record matched
  line matched
file matched
```

The ANTLR grammar so far can be downloaded here: csv1.g [files/translate/csv1.g]. It is quite obvious from the output that the parser is performing as it should. To produce a HTML table, one only has to change the output statements so that instead of the parser outputting "file called", or whatever, it outputs "<table>" instead or whatever the HTML equivalent should be. The parser, modified to output HTML statements is shown below:

```
class CSVParser extends Parser;
options { k=2; }
file   {System.out.println("<table align=\"center\" border=\"1\">");}
       : ( line (NEWLINE line)* (NEWLINE)? EOF)
       {System.out.println("</table>");}
       ;

line   {System.out.println("  <tr>");}
       : ( (record)+ )
       {System.out.println("  </tr>");}
       ;

record {System.out.print("    <td>");}
       : ( (r:RECORD) (COMMA)? )
       {System.out.print(r.getText());
        System.out.println("</td>");}
       ;
```

The output produced when processing the same file as before is:

```
<table align="center" border="1">
  <tr>
    <td>David Sprocket</td>
    <td>89</td>
  <tr/>
  <tr>
```

```
    <td>Cindy Brocket</td>
    <td>18</td>
  <tr/>
  <tr>
    <td>Michael Rocket</td>
    <td>33</td>
  <tr/>
</table>
```

Which is what was intended of the translator. Apart from perhaps to mess around with how the output should look all that is needed now is to output the rest of the HTML file. Generation of the rest of the HTML file could be done in the parser by adding more to the *file* rule but this would clutter the parser, instead let's designate this task to the class implementing this parser and lexer. A *main* method could be placed in the parser part of the grammar so that, when the parser is generated, it has a *main* method that can be executed. Alternatively a separate class could be created that has a *main* method which implements the translator. This example will use a separate class. Here is that separate class:

```
import java.io.*;
public class Main {
   public static void main(String args[]) {
      if(args.length==0) { error(); }

      FileInputStream fileInput = null;
      try {
         fileInput = new FileInputStream(args[0]);
      } catch(Exception e) { error(); }

      try {
         DataInputStream input = new DataInputStream(fileInput);

         CSVLexer csvLexer =   new CSVLexer(input);
         CSVParser csvParser = new CSVParser(csvLexer);
         csvParser.file();
      } catch(Exception e) { System.err.println(e.getMessage()); }
   }

   private static void error() {
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - -*");
      System.out.println("| USAGE:                  |");
      System.out.println("|   java Main inputfile |");
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - -*");
      System.exit(0);
   }
}
```

The program first checks that the one compulsory command line argument has been provided. If it has not the program prints an error and exits. If it has, the program creates a new FileInputStream from the file specified, if there are any errors an error is printed and the program exits. If there are no errors the program enters another *try* block where a *DataInputStream* is setup using the file, the lexer created with this stream, the parser created with this lexer and then the parsers *file* method is called. If there are any *Exception*s thrown, they are caught and the message that came along with it is printed.

1. The Main program used to run the translator can be downloaded here: Main.java [files/translate/Main.java].
2. The full grammar file can be downloaded here: csv2.g [files/translate/csv2.g].
3. Some test data can be downloaded here: test.txt [files/translate/test.txt]

At the moment the parser outputs to *stdout*, this is not very useful. It would be more useful if the parser returned a string of the HTML output so that the calling class could do whatever it wanted with it, such as outputting it to a file. The parser must be modified to return a string:

```
class CSVParser extends Parser;
options { k=2; }
file returns[String table = new String()]
       {String lineData; table+="<table align=\"center\" border=\"1\">\n"; }
       : ( lineData=line {table+=lineData;}
            (NEWLINE lineData=line {table+=lineData;} )*
               (NEWLINE)? EOF )
       {table+="</table>";}
       ;

line returns [String lineData = new String()]
       {String recordData; lineData+="  <tr>\n";}
       : ( (recordData=record {lineData+=recordData;})+ )
       {lineData+="  <tr/>\n";}
       ;
```

```
record returns [String recordData = new String()]
       {recordData+=("    <td>");}
       : ( (rec:RECORD) (COMMA)? )
       {recordData+=(rec.getText());
        recordData+="</td>\n";}
       ;
```

The full grammar can be downloaded here: csv3.g [files/translate/csv3.g].

Within *file*: Immediately the opening *table* element is appended to the String *table* defined within the *file* rule. The first line is matched, and the value returned assigned to the variable *lineData* this is appended to *table*. Zero or more lines are matched, and in the process, the values returned from the call to *line* to match each line are assigned to the variable *lineData* which is appended to *table*.

```
line returns [String lineData = new String()]
       {String recordData; lineData+="  <tr>\n";}
       : ( (recordData=record {lineData+=recordData;})+ )
       {lineData+="  <tr/>\n";}
       ;
```

Within *line*: Immediately the opening *<tr>* element is appended to the *lineData* variable, one or more records are matched, and each time the value returned from the call to *record* to match a record is assigned to the variable *recordData* which is then appended to *lineData*.

```
record returns [String recordData = new String()]
       {recordData+=("    <td>");}
       : ( (rec:RECORD) (COMMA)? )
       {recordData+=(rec.getText());
        recordData+="</td>\n";}
       ;
```

Within *record*: Immediately the opening *td* element is appended to the string *recordData*. A *RECORD* token is matched and assigned to the *rec* variable, the textual content of this is then appended to *recordData* and the closing *</td>* appended to *recordData*. The completed record in the form of *recordData* is returned to *line*.

line receives *recordData* and this is used in the construction of *lineData*, when all the records have been matched for a line the closing *</tr>* is appended to *lineData* and *lineData* returned.

file receives *lineData* and this is used in the construction of *table*, when all the lines have been matched for the file the closing */<table>* tag is appended to *table* and *table* returned to the class that instantiated the parser.

Here is the new *main* method containing class:

```
import java.io.*;
public class CSVhtml {
   public static void main(String args[]) {
      if(args.length!=2) { error(); }

      FileInputStream fileInput = null;
      DataOutputStream fileOutput = null;
      try {
         fileInput = new FileInputStream(args[0]);
         fileOutput = new DataOutputStream(new FileOutputStream(args[1]));
      } catch(Exception e) { error2(); }

      try {
         DataInputStream input = new DataInputStream(fileInput);

         CSVLexer csvLexer =   new CSVLexer(input);
         CSVParser csvParser = new CSVParser(csvLexer);
         String p ="";
         p ="<?xml version=\"1.0\" encoding=\"utf-8\"?>\n";
         p+="<!DOCTYPE html\n";
         p+="PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\"\n";
         p+="\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\n";
         p+="<html>\n";
         p+="  <head>\n";
         p+="    <title>A Table</title>\n";
         p+="     <meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\"/>\n";
         p+="  </head>\n";
         p+="  <body>\n";
         p+= csvParser.file();
         p+="  </body>\n";
         p+="</html>";
         fileOutput.writeBytes(p);
         fileOutput.close();
      } catch(Exception e) { System.err.println(e.getMessage()); }
   }
```

```
   private static void error() {
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*");
      System.out.println("| USAGE:                                    |");
      System.out.println("|   java CSVhtml inputfile outputfile |");
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*");
      System.exit(0);
   }

   private static void error2() {
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*");
      System.out.println("| You must specify a valid inputfile |");
      System.out.println("*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*");
      System.exit(0);
   }
}
```

The program shown above can be downloaded here: CSVHTML.java [files/translate/CSVHTML.java]. The program works like this: It is first checked that the command-line arguments are present, if they are, a *DataInputStream* is created for the inputfile which is specified by the first command-line argument and a *DataOutputStream* created for the outputfile which is specified by the second command-line argument. The lexers and parsers are created. A string is created and the HTML prologue appended. The string returned from calling the parsers *file* method is appended to this string. The HTML closure is appended to the string and the the string is output to the file supplied.

After the grammar is converted to a lexer, parsed by ANTLR and everything compiled, the program can be executed like this:

```
java CSVHTML inputfile outputfile
```

The table produced from executing the command:

```
java CSVHTML test.txt output.html
```

Looks, under a certain proprietary web-browser, like this:

Figure 10. `test.txt` expressed as a table

STUDENT ID	NAME	DATE OF BIRTH
129384	Davy Jones	03/04/81
328649	Clare Manstead	30/11/81
237090	Richard Stoppit	22/06/82
523343	Brian Hardwick	15/11/81
908423	Sally Brush	06/06/81
328453	Elisa Strudel	12/09/82
883632	Peter Smith	03/05/83
542033	Ryan Alcot	04/12/80

11. Snippets From Behind The Scenes

When the lexer tokenizes the input stream, each token encountered is catagorized into the type of token it is, such as a *NEWLINE* token. A table of these token types is created and each token type is represented by an integer. The integers 1-3 are special in that they denote predefined token types, user defined tokens are assigned an integer to represent them starting from 4. The integers are mapped to human readable identifiers in a token types file generated by ANTLR, for example:

```
// $ANTLR 2.7.1: "csv.g" -> "CSVParser.java"$

public interface CSVParserTokenTypes {
  int EOF = 1;
  int NULL_TREE_LOOKAHEAD = 3;
  int NEWLINE = 4;
  int RECORD = 5;
```

```
  int COMMA = 6;
  int WS = 7;
}
```

There is a class called *TokenBuffer* whose job it is to buffer the tokens provided by the lexer. It contains a method called *LA* which has one parameter, an integer, which determines the token in the token buffer to return, for example *LA(1)* would return the integer value of the next token in the TokenBuffer. The parser uses a series of calls to *LA* to match the rules it implements, for example:

```
// Note, I have cleaned this up a little, ANTLR generates things like
// { {instructions} {instructions} } which can be represented as
// { instructions instructions }
public final void file() throws RecognitionException, TokenStreamException {
   try {      // for error handling
      System.out.println("file called");
      int _cnt3=0;
      _loop3:
      do {
         if ((LA(1)==RECORD)) {
            line();
         } else {
            if ( _cnt3>=1 ) { break _loop3; }
            else {throw new NoViableAltException(LT(1), getFilename());}
         }
         _cnt3++;
      } while (true);
   } catch (RecognitionException ex) {

   reportError(ex);
   consume();
   consumeUntil(_tokenSet_0);
}
```

TokenBuffer provides tokens via *LT* and tokens via *LA*. TokenBuffer gets it's tokens for buffering by calling, the method *nextToken* which is defined in the Lexer. The method *nextToken* in the Lexer generated for the CSV translator looks like this:

```
// Cleaned up a little by me
public Token nextToken() throws TokenStreamException {
   Token theRetToken=null;
tryAgain:
   for (;;) {
      Token _token = null;
      int _ttype = Token.INVALID_TYPE;
      resetText();
      try {   // for char stream error handling
         try {   // for lexical error handling
            switch ( LA(1)) {

               case ',': {
                  mCOMMA(true);
                  theRetToken=_returnToken;
                  break;
               }

               case '\n':  case '\r': {
                  mNEWLINE(true);
                  theRetToken=_returnToken;
                  break;
               }

               case '\t':  case ' ': {
                  mWS(true);
                  theRetToken=_returnToken;
                  break;
               }

               default:
               if ((_tokenSet_0.member(LA(1)))) {
                  mRECORD(true);
                  theRetToken=_returnToken;
               } else {
                  if (LA(1)==EOF_CHAR) {uponEOF(); _returnToken = makeToken(Token.EOF_TYPE);}
                  else {throw new NoViableAltForCharException((char)LA(1), getFilename(), getLine());}
               }
            }
            if ( _returnToken==null ) continue tryAgain; // found SKIP token

            _ttype = _returnToken.getType();
            _ttype = testLiteralsTable(_ttype);
            _returnToken.setType(_ttype);
            return _returnToken;
         } catch (RecognitionException e) {
```

```
            throw new TokenStreamRecognitionException(e);
         }
      } catch (CharStreamException cse) {
         if ( cse instanceof CharStreamIOException ) {
            throw new TokenStreamIOException(((CharStreamIOException)cse).io);
         } else {
            throw new TokenStreamException(cse.getMessage());
         }
      }
   }
}
```

Notice the bit that says:

```
if (LA(1)==EOF_CHAR) {uponEOF(); _returnToken = makeToken(Token.EOF_TYPE);}
else {throw new NoViableAltForCharException((char)LA(1), getFilename(), getLine());}
```

This says if the token found is an *EOF_CHAR*, call the *uponEOF* method and assign a new *EOF_TYPE* token to *_returnToken*, which is returned later in this method.

The code below shows *rec* being assigned the Token returned from *LT*, later, the *getText()* method is invoked on *rec* to get the tokens textual content.

```
   {
   rec = LT(1);
   match(RECORD);
   }
   .
   .
   .
   recordData+=(rec.getText());
   recordData+="</td>\n";
```

12. Thanks

Thanks go to Bogdan Mitu for showing me the way with the translator example *file* rule and Ric Klaren for showing me how blindingly simple it was to do the nested return statements in the translator example.

13. References (And links you may find useful)

- http://www.antlr.org/book/index.html

  Practical Computer Language Recognition and Translation
  A guide for building source-to-source translators with ANTLR and Java.

  Copyright 1999 Terence Parr

  Updated 2/1/99

- http://www.antlr.org/article/list
  ANTLR articles page - lots of interesting things.
- http://www.antlr.org/doc/getting-started.html
  Getting Started with ANTLR.
- http://javadude.com/articles/antlrtut/
  An ANTLR Tutorial by Scott Stanchfield
- http://topaz.cs.byu.edu/text/html/Textbook/
  Compiler Theory And Design
- The ANTLR Reference Manual
  Comes included with the ANTLR installation