ANTLR - Introduction

(ANother Tool for Language Recognition)

Vertaalerbouw **HC4**

VB HC4

Arend Rensink
kamer: Zilverling 5090
telefoon: 4862
email: rensink@cs.utwente.nl

Original design: Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools

What have you seen last time?

- AST implementation
  - Construction during parsing
- Scoping
  - Blocks
  - Symbol tables
- Type checking
  - Type rules
  - Visitor pattern

Overview of Lecture 4

www.antlr.org

ANTLR v3

- Introduction
- ANTLR 3.4 by Example
  - Calc – a simple calculator language
- Some ANTLR grammar patterns

ANTLR – Introduction (1)

www.antlr.org

- ANTLR
  - input: language descriptions using EBNF grammar
  - output: recognizer for the language
- ANTLR can build recognizers for three kinds of input:
  - character streams (i.e. by generating a scanner)
  - token streams (i.e. by generating a parser)
  - node streams (i.e. by generating a tree walker)

  ANTLR uses the same syntax for all its recognizer descriptions.

© Theo Ruys, Arend Rensink

ANTLR – Introduction (2)

www.antlr.org

- ANTLR generates (predictive) LL(k) or LL(*) recognizers
  - Computes first, follow and lookahead sets
  - Verifies syntax conditions (e.g. LL(k) test)
  - Generated scanner/lexer is a predictive recursive-descent recognizer and not a finite automaton.
- Terminology
  - lexer = scanner, lexical analyser, tokenizer
  - parser = syntactical analyser
  - tree parser = tree walker
- Other well-known compiler generators
  - scanners: lex/flex, JFlex
  - parsers: yacc/Bison, JCup, javaCC, sableCC, SLADE

© Theo Ruys, Arend Rensink

ANTLR – Introduction (3)

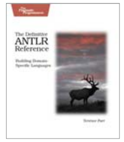
www.antlr.org

- Material on ANTLR:
  - See ANTLR's website.
  - There is a wealth of information on ANTLR. Unfortunately, the documentation is not very well structured and might be overwhelming for beginners.

    Spend some time browsing the documentation to get an overview of what is available.
  - Yahoo group: antlr-interest (also as mailing-list)

    Active community: quite some traffic!
- Book:

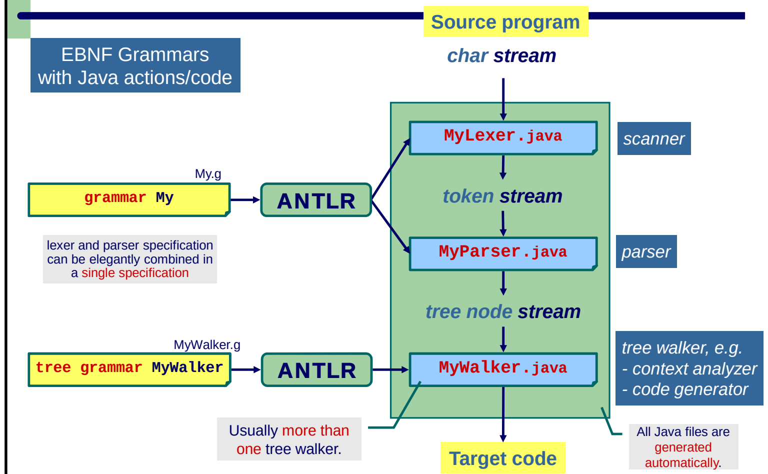
  Terence Parr.
  The Definitive ANTLR Reference.
  Pragmatic Bookshelf, 2007.



PDF freely available

© Theo Ruys, Arend Rensink

ANTLR – overview



© Theo Ruys, Arend Rensink

ANTLR – input file structure

*.g

```
[gtype] grammar FooBar;
options {
  options for entire grammar file
}

tokens {
  extra token definitions
}

@header {
  will be copied to the generated Java file(s)
}

@rulecatch {
  error handling: how to deal with exceptions?
}

@members {
  optional class definitions: instance variables, methods
}

rulename : all rules for FooBar
```

gtype may be empty or **tree**.

A single .g file can specify a Lexer + Parser, or a TreeParser.

e.g. **imports**

also: **@lexer::header** and **@lexer::members**

ANTLR – rule structure

ANTLR generates a recursive descent recognizer: for each rule, ANTLR will generate a Java method.

```
rulename [args] returns [T val]
  options { local options }
    : alternative_1
    | alternative_2
    | ...
    | alternative_n
    ;
```

optional, used for passing information around

An alternative is an EBNF expression containing:

- rulenames
- TOKENs
- EBNF operators
- Java code in braces

+ optional code sections to insert at start of end of method

```
@init  { ... }
@after { ... }
```

EBNF operators

A|B	A or B
A*	zero or more A's
A+	one or more A's
A?	an optional A

When using EBNF operators in ANTLR: use parentheses to enclose more than one symbol..

Example

```
expr    : operand (PLUS operand)*
        ;
operand : LPAREN expr RPAREN
        | NUMBER
        ;
```


Running ANTLR

- ANTLR is a Java program:

```
java org.antlr.Tool file.g
```

may contain specifications for a lexer and parser, or a treewalker

The ANTLR jar-file should be in the CLASSPATH of course.

- By default Java generates .java files which have to be compiled to an Java application.
- There also exist several ANTLR GUI Development Environments:
  - ANTLRv3 IDE (for Eclipse) http://antlrv3ide.sourceforge.net/
  - ANTLRWorks http://www.antlr.org/works/

Calc – Language (1)

Will be extended upon in the laboratory of week 3 and 4

- Calc: simple calculator language
  - declarations
    - only integer variables
    - must all come before statements
  - statements
    - assignment to variables
    - printing of expressions

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```


Calc – Language (2)

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

- EBNF for Calc

program	::=	declarations statements EOF
declarations	::=	(declaration SEMICOLON)*
declaration	::=	VAR IDENTIFIER COLON type
statements	::=	(statement SEMICOLON)+
statement	::=	assignment
		| printStatement
assignment	::=	lvalue BECOMES expr
printStatement	::=	PRINT LPAREN expr RPAREN
lvalue	::=	IDENTIFIER
expr	::=	operand ((PLUS | MINUS) operand)*
operand	::=	IDENTIFIER
		| NUMBER
		| LPAREN expr RPAREN
type	::=	INTEGER

All terminals are written as UPPERCASE symbols.

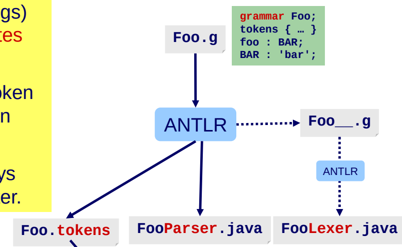
Calc compiler – overview

- We let ANTLR generate four recognizers:
  - CalcLexer *(extends Lexer)*
    - translates a stream of characters to stream of tokens
  - CalcParser *(extends Parser)*
    - translates a stream of tokens to an stream of tree nodes
  - CalcChecker *(extends TreeParser)*
    - reads the stream of tree nodes (i.e. the AST) and checks whether the context constraints are obeyed
  - CalcInterpreter *(extends TreeParser)*
    - reads the stream of tree nodes (i.e. the AST) and executes the program

ANTLR - Parser & Lexer

- A lexer and parser are closely related. A lexer generates tokens which are consumed by a parser.
- In ANTLR 3.x, the lexer and parser can be combined elegantly into a single grammar specification.
  - ANTLR takes care of splitting the two specifications.

- Literals (i.e. character strings) are enclosed in single quotes (e.g. `'bar'`).
- Lexer non-terminals (i.e. token names) always start with an UPPERCASE letter.
- Parser non-terminals always start with an lowercase letter.



Will be imported by tree parsers.

Calc – Parser & Lexer (1)

```
grammar Calc;

options {
  k = 1;
  language = Java;
  output = AST;
}

tokens {
    PLUS       =  '+'  ;
    MINUS      =  '-'  ;
    BECOMES    =  ':=' ;
    COLON      =  ':'  ;
    SEMICOLON  =  ';'  ;
    LPAREN     =  '('  ;
    RPAREN     =  ')'  ;

    // keywords
    PROGRAM    =  'program' ;
    VAR        =  'var'     ;
    PRINT      =  'print'   ;
    INTEGER    =  'integer' ;
}
```

This is a combined specification (not prefixed by lexer, parser or tree).

amount of lookahead, disables LL(*)

Target language is Java.

build an AST

token definitions (literals)

tokens always start with an uppercase letter and specify the text for a token

Calc – Parser & Lexer (3)

lexer specific rules

```
IDENTIFIER :   LETTER (LETTER | DIGIT)*
           ;

NUMBER     :   DIGIT+
           ;

COMMENT    :   '//' .* '\n'
                 { $channel=HIDDEN; }
           ;

WS         :   (' ' | '\t' | '\f' | '\r' | '\n')+
                 { $channel=HIDDEN; }
           ;

fragment DIGIT  :   ('0'..'9') ;
fragment LOWER  :   ('a'..'z') ;
fragment UPPER  :   ('A'..'Z') ;
fragment LETTER :   LOWER | UPPER ;
```

"`.*`" matches everything except the character that follows it (i.e. `'\n'`).

There are multiple token channels. The parser reads from the DEFAULT channel. By setting a token's channel to HIDDEN it will be ignored by the parser.

shorthand for (the complete) `'a'|'b'|'c'| ...|'y'|'z'`

fragment lexer rules can be used by other lexer rules, but do not return tokens by themselves

Calc – Parser & Lexer (2)

First only recognising, no AST construction yet.

parser specific rules

```
program       :   declarations statements EOF
              ;
declarations  :   (declaration SEMICOLON)*
              ;
statements    :   (statement SEMICOLON)+
              ;
declaration   :   VAR IDENTIFIER COLON type
              ;
statement     :   assignment
              |   print
              ;
assignment    :   lvalue BECOMES expr
              ;
print         :   PRINT LPAREN expr RPAREN
              ;
lvalue        :   IDENTIFIER
              ;
expr          :   operand ((PLUS | MINUS) operand)*
              ;
operand       :   IDENTIFIER
              |   NUMBER
              |   LPAREN expr RPAREN
              ;
type          :   INTEGER
              ;
```

special "end-of-file" token

parser rules start with a lowercase letter

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

In this example, all tokens are explicitly named (as UPPERCASE tokens). It is also possible to use literals in the parser specification.
For example:

```
print : 'print' '(' expr ')'
expr  : operand (('+' | '-') operand)*
```


Calc – Parser & Lexer (4)

The Parser does not only recognize the language, it should also build the AST.

parser building the AST

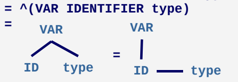
```
program       :   declarations statements EOF
                    -> ^(PROGRAM declarations statements)
              ;
declarations  :   (declaration SEMICOLON!)*
              ;
statements    :   (statement SEMICOLON!)+
              ;
declaration   :   VAR^ IDENTIFIER COLON! type
              ;
statement     :   assignment
              |   print
              ;
assignment    :   lvalue BECOMES^ expr
              ;
print         :   PRINT^ LPAREN! expr RPAREN!
              ;
lvalue        :   IDENTIFIER
              ;
expr          :   operand ((PLUS^ | MINUS^) operand)*
              ;
operand       :   IDENTIFIER
              |   NUMBER
              |   LPAREN! expr RPAREN!
              ;
type          :   INTEGER
              ;
```

Imaginary token used as the root node.

Annotations for building AST nodes	
T^	make T the root of this (sub)rule
T!	discard T
-> ^(…)	tree construction for a rule

For example:

```
  VAR^ IDENTIFIER COLON! type
= ^(VAR IDENTIFIER type)
=
```



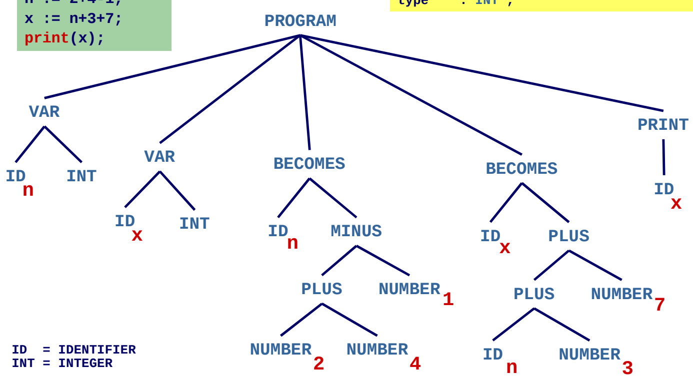
first child, next sibling notation

due to rule of operand this builds:
^(PLUS expr expr)

AST Tree (Example)

```
program : decls stats EOF
            -> ^(PROGRAM decls stats);
decls   : (decl SEMICOLON!)*
stats   : (stat SEMICOLON!)+ ;
decl    : VAR^ ID COLON! type ;
stat    : assign | print ;
assign  : lvalue BECOMES^ expr ;
print   : PRINT^ LPAREN! expr RPAREN! ;
lvalue  : ID ;
expr    : oper ((PLUS^ | MINUS^) oper)* ;
oper    : ID | NUM | LPAREN! expr RPAREN! ;
type    : INT ;
```

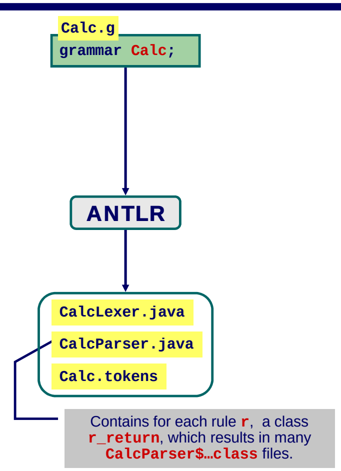
```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```



ID = IDENTIFIER
INT = INTEGER

Calc - generated Java files

DEMO



Calc – visualizing the AST (1)

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

    CalcLexer lexer = new CalcLexer(
                            new ANTLRInputStream(System.in));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    CalcParser parser = new CalcParser(tokens);

    CalcParser.program_return result = parser.program();
    CommonTree tree = (CommonTree) result.getTree();

    // show S-Expression respresentation of the AST
    String s = tree.toStringTree();
    System.out.println(s);

    // print the AST as DOT specification
    DOTTreeGenerator gen = new DOTTreeGenerator();
    StringTemplate st = gen.toDOT(tree);
    System.out.println(st);
}
```

lexer

parser

-ast

-dot

A lexer gets an ANTLR stream as input.

The parser gets the lexer's output tokens.

Call the start symbol to start parsing.

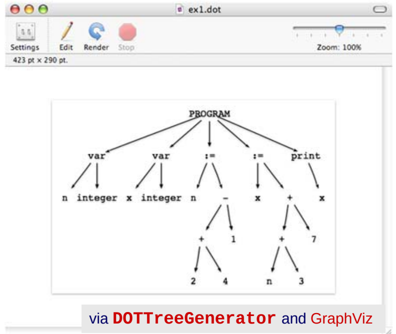
.dot files can be visualized using the GraphViz program: http://www.graphviz.org/

`DOTTreeGenerator` is defined in package `org.antlr.stringtemplate`

Calc – visualizing the AST (2)

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```

via `tree.toStringTree()`



via `DOTTreeGenerator` and GraphViz

Calc Parser – Java code (1)

```
public class CalcParser extends Parser {
   ...
  public final program_return program() throws RecognitionException {
      program_return retval = new program_return();
      ...
      try {
          // Calc.g:44:9: declarations statements EOF
          {
          pushFollow(FOLLOW_declarations_in_program412);
          declarations1=declarations();
          _fsp--;

          stream_declarations.add(declarations1.getTree());
          pushFollow(FOLLOW_statements_in_program414);
          statements2=statements();
          _fsp--;

          stream_statements.add(statements2.getTree());
          EOF3=(Token)input.LT(1);
          match(input,EOF,FOLLOW_EOF_in_program416);
          stream_EOF.add(EOF3);
          ...
          }
      }
      catch (RecognitionException re) {
          reportError(re);
          recover(input,re);
      } ...
      return retval;
  }
```

Most code that builds the AST is omitted!

```
program
  : declarations statements EOF!
  ;
```


Calc Parser – Java code (2)

```
public final declarations_return declarations() throws RecognitionException {
    declarations_return retval = new declarations_return();
    ...
    try {
        ...
        loop1:
        do {
            int alt1=2;
            int LA1_0 = input.LA(1);

            if ( (LA1_0==VAR) )
                alt1=1;

            switch (alt1) {
                case 1 :
                    {
                    pushFollow(FOLLOW_declaration_in_declarations463);
                    declaration4=declaration();
                    ...
                    match(input,SEMICOLON,FOLLOW_SEMICOLON_in_declarations465);
                    }
                    break;
                default :
                    break loop1;
            }
        } while (true);
    } catch (RecognitionException re) {
        ...
    return retval;
}
```

LA(1) - current lookahead Token.

```
declarations
  : (declaration SEMICOLON!)*
  ;
```

Calc – Tree walker

```
program : decls stats EOF -> ^(PROGRAM decls stats);
decls   : (decl SEMICOLON!)*
stats   : (stat SEMICOLON!)+ ;
decl    : VAR^ IDENTIFIER COLON! type ;
stat    : assign | print ;
assign  : lvalue BECOMES^ expr ;
print   : PRINT^ LPAREN! expr RPAREN! ;
lvalue  : ID ;
expr    : operand ((PLUS^ | MINUS^) operand)* ;
operand : ID | NUM | LPAREN! expr RPAREN! ;
type    : INT ;
```

```
tree grammar CalcTreeWalker;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}

program     :   ^(PROGRAM (declaration | statement)+)
            ;

declaration :   ^(VAR IDENTIFIER type)
            ;

statement   :   ^(BECOMES IDENTIFIER expr)
            |   ^(PRINT expr)
            ;

expr        :   operand
            |   ^(PLUS expr expr)
            |   ^(MINUS expr expr)
            ;

operand     :   IDENTIFIER | NUMBER ;
type        :   INTEGER ;
```

This is a specification of a tree walker.

Import tokens from **Calc.tokens**.

The AST nodes are of type **CommonTree**.

The AST has a root node **PROGRAM** with many (declaration or statement) children.

Match a tree whose root is a PLUS token with two children that match the expr rule.

This tree walker does not do anything (yet). Note the conciseness of the grammar and the correspondence with the "abstract syntax" of the language Calc.

Calc – Checker (1)

CalcChecker checks the context rules of the language:
- each identifier can be declared only once
- identifiers that are used must have been declared.

```
tree grammar CalcChecker;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}

@header {
import java.util.Set;
import java.util.HashSet;
}

@rulecatch {
catch (RecognitionException e) {
  throw e;
}
}

@members {
   private Set<String> idset = new HashSet<String>();
   public boolean isDeclared(String s) {
       return idset.contains(s);
   }
   public void declare(String s) {
       idset.add(s);
   }
}
```

@header: code block which is copied verbatim to the beginning of CalcChecker.java.

@rulecatch: specify your own error handler. Here: no error handler; exceptions are propagated to the method calling this checker.

@members: code block which is copied verbatim to the class definition of CalcChecker.java.

The Calc language uses a monolithic block structure. For checking the scope rules we can use a **Set**.

The methods **isDeclared** and **declare** become methods of the class **CalcChecker**.

Calc – Checker (2)

```
program
    :   ^(PROGRAM (declaration | statement)+)
    ;

declaration
    :   ^(VAR id=IDENTIFIER type)
        { if (isDeclared($id.getText()))
              throw new CalcException($id.getText() +
                        " is already declared");
          else
              declare($id.getText());
        }
    ;

statement
    :   ^(BECOMES id=IDENTIFIER expr)
        { if (!isDeclared($id.text))
              throw new CalcException($id.text +
                        " is used but not declared");
        }

    |   ^(PRINT expr)
    ;
...
```

With **name=NODE** we can refer to the AST node using **name** ...

... and get its String representation.

Within Java code, the ANTLR variables are (usually) prefixed with $.

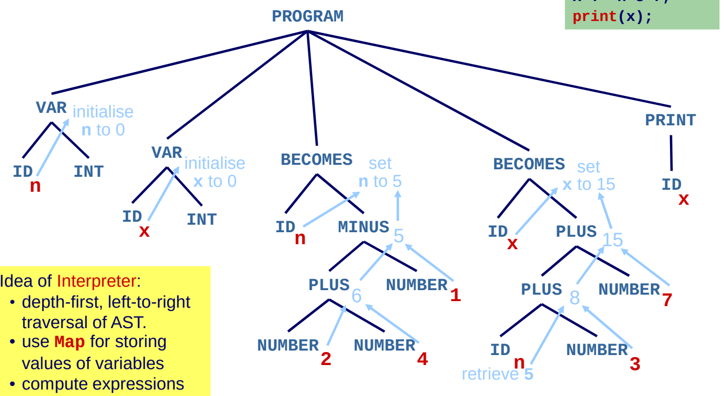
Java code block which is copied verbatim to the parse method of '**declaration**' in CalcChecker.java.

... or use the attribute **text**.

CalcException is an user-defined Exception (subclass of **org.antlr.runtime RecognitionException**) to express some problem in the input.

Calc – Interpreter (1)

```
// ex1.calc
var n: integer;
var x: integer;
n := 2+4-1;
x := n+3+7;
print(x);
```



Idea of Interpreter:
- depth-first, left-to-right traversal of AST.
- use **Map** for storing values of variables
- compute expressions bottom up

Calc – Interpreter (2)

The structure of the tree grammar CalcInterpreter is the same as for CalcChecker

```
tree grammar CalcInterpreter;

options {
    tokenVocab = Calc;
    ASTLabelType = CommonTree;
}
@header {
import java.util.Map;
import java.util.HashMap;
}
@members {
    Map<String,Integer> store = new HashMap<String,Integer>();
}

program      :   ^(PROGRAM (declaration | statement)+)
             ;

declaration  :   ^(VAR id=IDENTIFIER type)
                 { store.put($id.text, 0); }
             ;

...
```

Idea of Interpreter:
- depth-first, left-to-right traversal of AST.
- use **Map** for storing values of variables
- compute expressions bottom up

To store the values of the variables.

Initialized on 0.

Calc – Interpreter (2)

```
statement
    :   ^(BECOMES id=IDENTIFIER v=expr)
        { store.put($id.text, $v); }

    |   ^(PRINT v=expr)
        { System.out.println("" + $v); }
    ;

expr returns [int val = 0]
    :   z=operand                 { val = z;     }
    |   ^(PLUS  x=expr y=expr)    { val = x + y; }
    |   ^(MINUS x=expr y=expr)    { val = x - y; }
    ;

operand returns [int val = 0]
    :   id=IDENTIFIER { val = store.get($id.text); }
    |   n=NUMBER      { val = Integer.parseInt($n.text); }
    ;
```

The rule expr returns a value.

The value returned by expr is put into the store for id.

ANTLR deduces from the context the types of the variables: **id** is a **CommonTree**, **v** is an **int**.

A rule can return a value: **rulename returns [T x]**
The type of the return value is **T** and the value returned is the value of **x** at the end of the rule.

Note that it is also possible to pass arguments to a rule.

Get the value of **IDENTIFIER** out of the store.

Parse the string representation of the **NUMBER**.

Calc – driver

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

    CalcLexer  lex  = new CalcLexer(
                          new ANTLRInputStream(System.in));
    CommonTokenStream tokens = new CommonTokenStream(lex);
    CalcParser parser = new CalcParser(tokens);

    CalcParser.program_result result = parser.program();
    CommonTree tree = (CommonTree) result.getTree();

    CommonTreeNodeStream nodes1 = new CommonTreeNodeStream(tree);
    CalcChecker checker = new CalcChecker(nodes1);
    checker.program();

    CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(tree);
    CalcInterpreter interpreter = new CalcInterpreter(nodes2);
    interpreter.program();
}
```

lexer

parser

checker

inter-preter

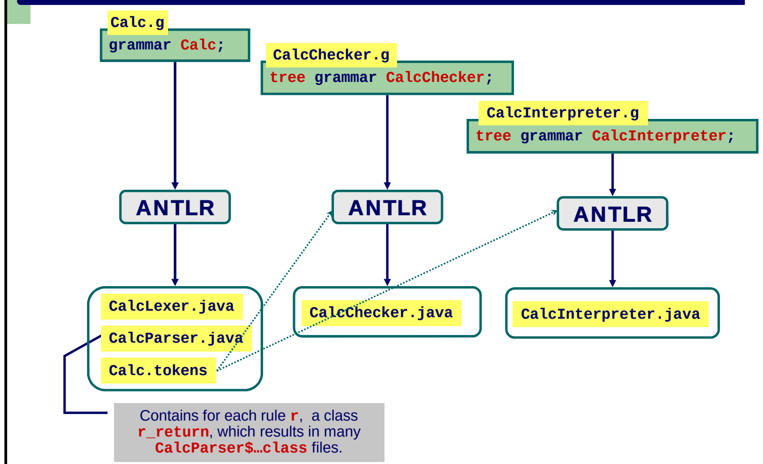
A lexer gets an ANTLR stream as input.

The parser gets the lexer's output tokens.

Call the start symbol to start parsing.

The recognition methods may all throw Exceptions (e.g. **RecognitionException**, **TokenStreamException**); These have to be caught in **main**-method. See **Calc.java**.

Calc - generated Java files



Advantages ANTLR

- With ANTLR you can specify your compiler and let ANTLR do the hard work of generating the compiler.

  But the generated Java code is similar to what you would write manually: it is possible (and easy!) to read and debug Java files generated by ANTLR.

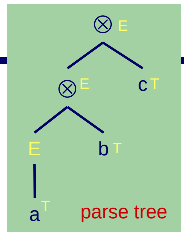
- The syntax for specifying scanners, parsers and tree walkers is the same.
- ANTLR can generate recognizers for many programming languages (e.g. Java, C#, Python, (Objective) C, etc.)
- ANTLR is well supported and has an active user community.

Some ANTLR Tips

- left associative
- right associative
- operator precedence
- dangling-else

Second lecture on ANTLR will discuss some more advanced ANTLR Tips and Techniques.

Left associative



- Left associative operator ⊗ :

  $a \otimes b \otimes c \equiv (a \otimes b) \otimes c$

- Production rule:

  E ::= E ⊗ T | T

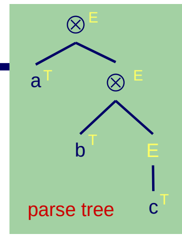
  which can be written (by eliminating left recursion) as

  E ::= X Y
  X ::= T
  Y ::= ⊗ T Y
  | empty

- or using EBNF:

  E ::= T (⊗ T)*

Right associative



- Right associative operator ⊗ :

  $a \otimes b \otimes c \equiv a \otimes (b \otimes c)$

- Production rule:

  E ::= T ⊗ E | T

  which can be written (using left factorisation) as

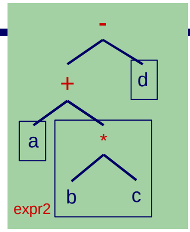
  E ::= T X
  X ::= ⊗ E
  | empty

- or using EBNF:

  E ::= T (⊗ E)?

Operator Precedence (1)

- Consider the following example
  a + b * c - d
- which should be parsed as
  (a + (b * c)) - d



- This means that the operator * has precedence over the operators + and -. This can be reflected in the grammar by making sure that * is *'closer to the operands'* than + and -.

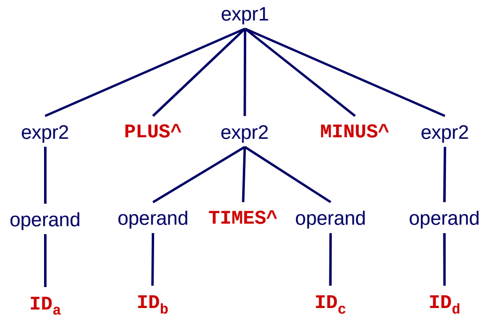
```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```


Operator Precedence (2)

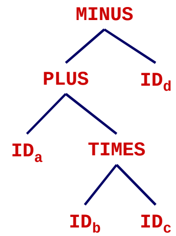
a + b * c - d

```
expr1   : expr2 ((PLUS^ | MINUS^) expr2)*
expr2   : operand (TIMES^ operand)*
operand : IDENTIFIER
```

parse tree:



constructed AST:



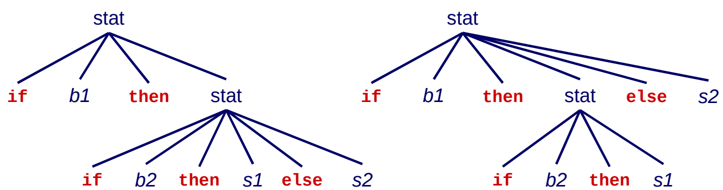
Greedy (1)

- Consider the classic if-then-else ambiguity (i.e., dangling else)

```
stat : 'if' expr 'then' stat ('else' stat)?
     | ...      ;
```

e.g. **if** *b1* **then** **if** *b2* **then** *s1* **else** *s2*

Two possible parse trees:



Greedy (2)

- So this ambiguity (which statement should the "else" be attached to) results in a parser nondeterminism. ANTLR warns you:

```
warning(200): Foo.g:12:33: Decision can match input
such as "'else'" using multiple alternatives: 1, 2
As a result, alternative(s) 2 were disabled for that
input
```

- If you make it clear to ANTLR that you want the subrule to match greedily, ANTLR will not generate the warning.

```
stat : 'if' expr 'then' stat
       (options {greedy=true;} : 'else' stat)?
     | ...      ;
```