LL (Top-Down) Parsing

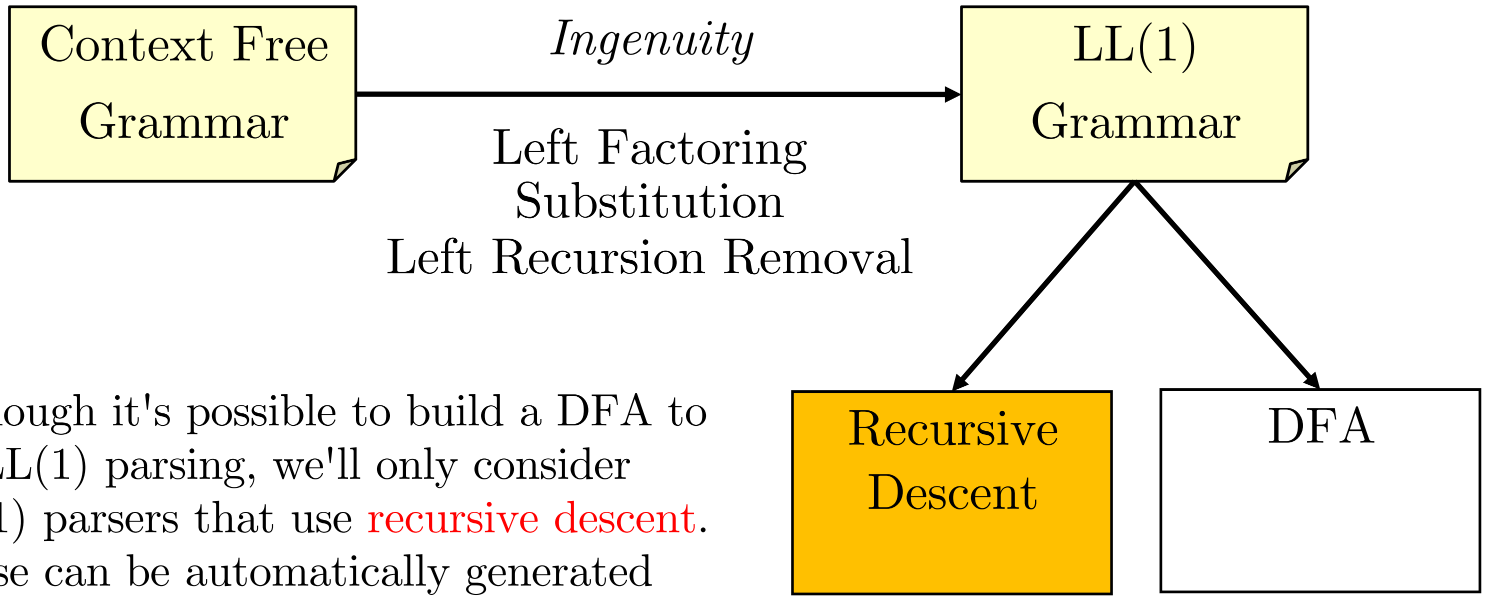
Tokens to AST

Naranker Dulay
n.dulay@imperial.ac.uk

https://www.doc.ic.ac.uk/~nd/compilers

Approach

A grammar is LL(k) if k-token lookahead is sufficient to choose between the alternatives of a rule when parsing. For LL(1) by looking at 1 token only.



Although it's possible to build a DFA to do LL(1) parsing, we'll only consider LL(1) parsers that use recursive descent. These can be automatically generated from the grammar (e.g. with ANTLR4) or hand-written.

LL(k) Grammars

More formally, a grammar is LL(1) if

(1) for each distinct pair of alternatives (α, β) of a rule (non-terminal) `A`, **FIRST(α)** and **FIRST(β)** are disjoint, e.g. for

$$A \rightarrow \alpha \mid \beta$$

the tokens (terminals) that could start $\boldsymbol{\alpha}$ are distinct from the tokens that could start $\boldsymbol{\beta}$

<u>**and**</u>

(2) for every rule `A`, if **FIRST(A)** contains ε then

FIRST(A) and **FOLLOW(A)** are disjoint.

i.e. the tokens that could start **A** are distinct from the tokens that can follow **A**.

Backus-Naur Form (BNF)

The rules for a context free grammar take the following form:

$A \rightarrow \alpha$ where A is a rule (non-terminal) and α is a possibly empty sequence of rules and and tokens (terminals). Note: A can be defined more than once.

When defining Programming Languages this form is sometimes referred to as **Backus-Naur Form (BNF)** after two of the early pioneers of programming language design. In this *basic* BNF form there is no special notation for Grouping, Repetition or Alternation - these constructs do not make the notation any more powerful.

We'll use `|` for alternation instead of writing multiple rules. If we need to use `|` as a terminal we'll quote it, e.g. `'|'`.

Extended BNF (EBNF)

Since repetitive and optional constructs are very common in programming language syntax many use an Extended form of BNF (EBNF) for writing context free grammars. EBNF shortcuts include:

$\{\alpha\}$	0 or more occurrences of α	(Repetition)
$[\alpha]$	0 or 1 occurrence of α	(Optional)
(α)	α. Useful for grouping alternatives $(\alpha \mid \beta \mid \delta)$	(Grouping)

BNF	EBNF
`Expr → Term1 '+' Term2` `Expr → Term1`	`Expr → Term1 ['+' Term2]`
`Seq → Seq ';' Stmt1` `Seq → Stmt2`	`Seq → Stmt2 {';' Stmt1}`

We'll use EBNF to simplify the transformation of context-free grammars into LL(1) grammars.

Recursive Descent Parser

A recursive descent parser for an LL(1) grammar consists of

- a set of **parse functions**, one for each rule **`A`** e.g. **`A()`**
  Note: the parse function should also return an AST for the rule.
- the current input token e.g. a variable **`token`**
- a token **match** and advance function, e.g.

```
def match(expected):
   if token == expected:
                 token = scanner.get_token()
   else error(‘‘unexpected token encountered...’’)
```

Grammar to Parse Functions

We'll convert the body of a rule according to the following mapping:

Rule (non-terminal) `A` to `A()`

Token (terminal) `T` to `match(T)`

In the following we translate to **match(A)**, `match(B)` if `A`, `B` are tokens

`A B` ⇨ `A(); B();`

`A|B` ⇨

```
if   token in FIRST(A):  A()
elif token in FIRST(B):  B()
-- FIRST(A) and FIRST(B): must be disjoint
```

`{A}` ⇨

```
while token in FIRST(A): A()
-- FIRST(A) must be disjoint with what follows {A}
```

`[A]` ⇨

```
if token in FIRST(A):    A()
-- FIRST(A) must be disjoint with what follows [A]
```

Note: a grammar that satisfies these constraints is an LL(1) grammar

Example

```
Statement       → IfStatement | BeginStatement | PrintStatement
IfStatement     → if Expr then Statement [else Statement]
BeginStatement  → begin Statement {';' Statement} end
PrintStatement  → print Expr
```

```python
def Statement():
    if token == IF:         IfStatement()
    elif token == BEGIN:    BeginStatement()
    elif token == PRINT:    PrintStatement()
    else error()
```

Example contd

```
Statement       → IfStatement | BeginStatement | PrintStatement
IfStatement     → if Expr then Statement [else Statement]
BeginStatement  → begin Statement {';' Statement} end
PrintStatement  → print Expr
```

```
def IfStatement():
    match(IF)
    Expr()
    match(THEN)
    Statement()
    if token == ELSE:
        match(ELSE)
        Statement()
```

```
def BeginStatement():
    match(BEGIN)
    Statement()
    while token == SEMICOLON:
        match(SEMICOLON)
        Statement()
    match(END)
```

```
def PrintStatement():
    match(PRINT)
    Expr()
```

Building the AST

```
Statement       → IfStatement | BeginStatement | PrintStatement
IfStatement     → if Expr then Statement [else Statement]
BeginStatement  → begin Statement {';' Statement} end
PrintStatement  → print Expr
```

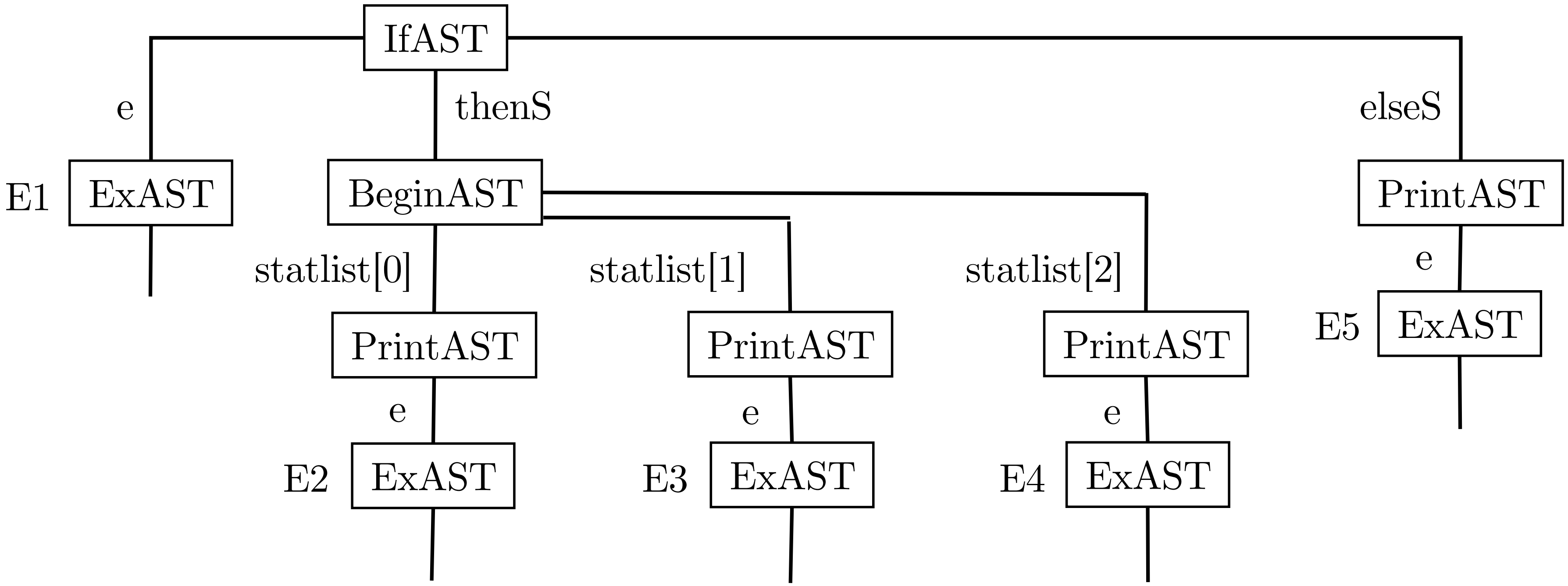
```
class StatementAST extends AST:
   Position p  # useful for error handling

class IfAST extends StatementAST:
   ExprAST e
   StatementAST thenS, elseS

class BeginAST extends StatementAST:
   StatementAST statlist[]     # list

class PrintAST extends StatementAST:
   ExprAST e
```

Define class hierarchies for the nodes in the ASTs and return a node from each parse function.

We could organise the AST to support traversal by later passes. For example, we could support the **visitor** design pattern. ANTLR4 supports both visitor and listener patterns.

Example

```
if E1 then
   begin print E2;  print E3; print E4 end
else
   print E5
```



Example

```
def Statement():
    if   token == IF:    return IfStatement()
    elif token == BEGIN: return BeginStatement()
    elif token == PRINT: return PrintStatement()
    else error()
```

```
def IfStatement():
    match(IF)
    e = Expr()
    match(THEN)
    thenS = Statement()
    if token == ELSE:
          match(ELSE)
          elseS = Statement()
    else:
          elseS = None
    return IfAST(e,thenS,elseS)
```

```
def BeginStatement():
  match(BEGIN)
  statlist = []
  statlist.append(Statement())
  while token == SEMICOLON:
     match (SEMICOLON)
     statlist.append(Statement())
  match(END)
  return BeginAST(statlist)
```

```
def PrintStatement():
   match(PRINT)
   return PrintAST(Expr())
```

CFG to LL(1)

In order to employ top-down parsing we need to ensure that our grammar is LL(1). Unfortunately transforming a non-LL(1) context-free grammar to LL(1) cannot be fully automated and often requires care and a little ingenuity. In particular we need to ensure that the transformed grammar has the same semantics as the original grammar and if possible retains the readability of the original.

The following 3 transformations are commonly tried:

(1) Left Factorisation

(2) Substitution

(3) Left Recursion Removal

Left recursion removal and substitution are normally carried out first then left factorisation.

Left Factorisation

If two or more grammatical alternatives share a common prefix we can factor out the common prefix using **Left Factorisation.**

Examples: In EBNF

A $\rightarrow$ B C | B D	$\Rightarrow$	A $\rightarrow$ B (C | D)
A $\rightarrow$ B C | B	$\Rightarrow$	A $\rightarrow$ B [C]

In basic BNF we would need to introduce an auxiliary rule, e.g.

A $\rightarrow$ B C | B D	$\Rightarrow$	A $\rightarrow$ B X X $\rightarrow$ C | D
A $\rightarrow$ B C | B	$\Rightarrow$	A $\rightarrow$ B X X $\rightarrow$ C | ε

Example

```
IfStatement → if Expr then Statement
            | if Expr then Statement else Statement
```

This grammar is not LL(1) because <u>*if* Expr *then* Statement</u> starts both alternatives. We can make it LL(1) by left factorising:

```
IfStatement → if Expr then Statement [ else Statement ]
```

In basic BNF the LL(1) grammar would be:

```
IfStatement → if Expr then Statement ElsePart
ElsePart    → else Statement | ε
```

Substitution

Substitution involves replacing a non-terminal `A` on the right-hand side of a rule by each of the alternatives of `A`. Substitution is useful when conflicts are "indirect" in order to make the conflict "direct" and hopefully amenable to left factoring. Substitution is also used to write clearer grammars. Example:

```
ForStatement → for ControlVar ':=' Expr Direction Expr do
                 Statement
ControlVar   → id
Direction    → to | downto
```

Although this grammar is LL(1) it makes sense to remove both `ControlVar` and `Direction` using substitution. The `ControlVar` rule seems to be defined purely as commentary - it serves no grammatical role otherwise. `Direction` is defined as a rule because the above grammar is given in BNF which doesn't support grouping of alternatives. By using substitution and EBNF the RHS becomes

```
for id ':=' Expr (to | downto) Expr do Statement
```

Example

```
Statement  → Assignment | ProcCall | pass
Assignment → id '=' Expr
ProcCall   → id '(' Expressions ')'
```

The grammar above is not LL(1) because ***id*** starts both `Assignment` and `ProcCall` and these are alternatives of `Statement`. In addition the grammar isn't in a form suitable for left factorisation. We can however bypass this little difficulty by substituting `Assignment` and `ProcCall` in `Statement` and then left factorising `Statement`.

```
Statement → id '=' Expr | id '(' Expressions ')' | pass

Statement → id ( '=' Expr | '(' Expressions ')' ) | pass
```

Although this form of the grammar is LL(1) it merges/obscures the clean separation in the original grammar. An LL(1) parser will need to handle this by re-introducing ASTnodes and parse functions for the alternatives and passing `id` to them in order to preserve the original form in later passes. The grammar is LL(2) - by looking at the next token, `'='` or `'('` we can decide what to do.

Left Recursion Removal

Grammars with left recursion cannot be LL(1). We'll cover the common case of **direct left recursion removal** which leaves the grammar largely intact. To eliminate a direct left recursive rule we'll re-write the rule using right-repetition (in BNF we can do the rewrite using a new auxiliary rule and right-recursion) – ANTLR 4 does this for you!. Handling of other recursive cases is more involved and often results in incomprehensible LL(1) grammars. Recall: Left recursion removal is unnecessary for LR parsers.

$$\mathtt{A} \rightarrow \mathtt{X} \mid \mathtt{A}\ \mathtt{Y} \quad \Rightarrow \quad \mathtt{A} \rightarrow \mathtt{X}\ \{\mathtt{Y}\}$$

More generally we have:

$$\mathtt{A} \rightarrow \mathtt{X}_1 \mid \mathtt{X}_2 \mid \ldots \mid \mathtt{X}_N \mid \mathtt{A}\ \mathtt{Y}_1 \mid \mathtt{A}\ \mathtt{Y}_2 \mid \ldots \mid \mathtt{A}\ \mathtt{Y}_N$$

$$\Rightarrow \mathtt{A} \rightarrow (\mathtt{X}_1 \mid \mathtt{X}_2 \mid \ldots \mid \mathtt{X}_N) \mid \mathtt{A}\ (\mathtt{Y}_1 \mid \mathtt{Y}_2 \mid \ldots \mid \mathtt{Y}_N)$$

$$\Rightarrow \mathtt{A} \rightarrow (\mathtt{X}_1 \mid \mathtt{X}_2 \mid \ldots \mid \mathtt{X}_N)\ \{\mathtt{Y}_1 \mid \mathtt{Y}_2 \mid \ldots \mid \mathtt{Y}_N\}$$

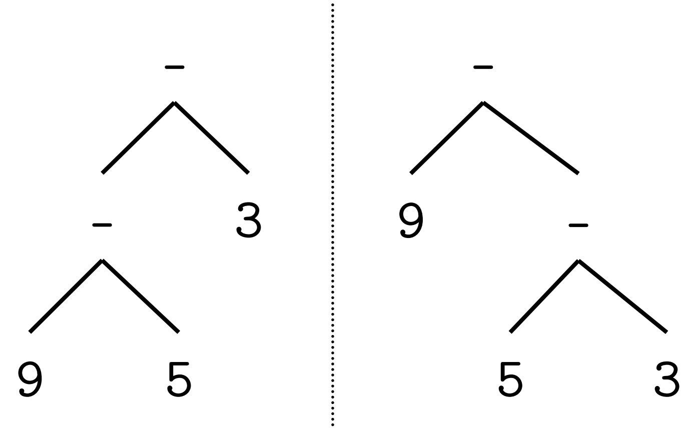
Example

```
Expr   → Expr AddOp Term | Term
Term   → Term MulOp Factor | Factor
Factor → ‘(‘ Expr ‘)’ | int
AddOp  → ‘+’ | ‘-’
MulOp  → ‘*’ | ‘/’
```

We can rewrite this as follows:

```
Expr   → Term {AddOp Term}
Term   → Factor {MulOp Factor}
Factor → ‘(‘ Expr ‘)’ | int
AddOp  → ‘+’ | ‘-’
MulOp  → ‘*’ | ‘/’
```

Note: Although the rewritten grammar is LL(1), it’s parse tree may no longer express the *left associatively* of subtraction and division. We need to ensure that when we construct the AST that we restore the associativity, e.g. for `9-5-3` we produce the left tree below:



Error Recovery

The response of a compiler to errors is a very important factor in its usefulness. Some compilers stop after producing a single error message. Most attempt some form of **recovery** and are capable of detecting further errors. Some compilers even attempt to **correct** syntactic errors in order to continue with semantic analysis and even code generation.

Requirements

- **Produce informative messages** for the user. Report the position of an error as accurately as possible. If the compiler is part of an Integrated Development Environment (IDE), highlight the region of each error in an editor window.
- If error recovery is attempted, **skip as little as possible** in order to parse as much of the remaining code as possible, **avoid infinite looping** in the recovery process, and avoid **spurious error messages** being generated as a result of the recovery.
- If error correction is attempted, ensure that the corrected program has the **same syntax tree and semantics** as the original (assuming we know what that is).

Error Recovery II

In **panic-mode error recovery** we provide each parse function with an extra parameter consisting of a **set of synchronising tokens** (the syncset).

As parsing proceeds additional tokens are added (if needed) to this set when calling other parse functions.

When an error occurs we skip ahead discarding tokens until one of the synchronising tokens is seen.

This technique works best when the compiler knows when not to "panic". This requires that the compiler writer choose the sets of synchronising tokens carefully when calling parse functions => **phrase-level error recovery**.

Common heuristics are to add all tokens in `FOLLOW(A)` to the syncset for a rule `A` and if `A` occurs in an "outer" construct to also add tokens from the `FIRST` set of the outer construct, e.g. expressions occur in statements, so we can add the `FIRST` set for statement to the syncset for expressions.

Resynchronising Methods

```
def match(expected):
   if token == expected:
        token = scanner.get_token()
   else error('unexpected token %s encountered at %s.
        Expecting %s', (token, token.position(), expected))
        # Note: error does not quit !!
```

```
def skipto(syncset):
   while token not in syncset and token != EOF:
        scanner.get_token()
```

```
def check(expectset,syncset,message):
   if token not in expectset:
        error(message)
        skipto(expectset union syncset)
```

Example

```
Statement    → IfStatement | BeginStatement | PrintStatement
IfStatement  → if Expr then Statement [else Statement]
```

```
def Statement(syncset):
   check ({IF,BEGIN,PRINT}, syncset, ‘‘Error ...’’)
   if   token == IF:    IfStatement(syncset)
   elif token == BEGIN: BeginStatement(syncset)
   elif token == PRINT: PrintStatement(syncset)
```

```
def IfStatement(syncset):
   match(IF)
   Expr(syncset union {THEN,ELSE})
   match(THEN)
   Statement(syncset union {ELSE})
   check({ELSE}, syncset, ‘‘Error ...’’)
   if token == ELSE:
         match(ELSE)
         Statement(syncset)
```

Example Contd

Statement → IfStatement | BeginStatement | PrintStatement

BeginStatement → ***begin*** Statement {';' Statement} ***end***

PrintStatement → ***print*** Expr

```
def BeginStatement(syncset):
    match(BEGIN)
    Statement(syncset union {SEMICOLON, END})
    while token == SEMICOLON:
        match(SEMICOLON)
        Statement(syncset union {SEMICOLON, END})
    match(END)
```

```
def PrintStatement(syncset):
    match(PRINT)
    Expr(syncset)
```

Summary

Top-down parsing is an elegant technique for constructing parsers. The normal approach is to manually transform a context-free grammar to an LL(1) grammar, and then to use either recursive descent, which has the virtue of all the parsing methods that we've considered, of being amenable to hand-coding; or to use an LL(1) pushdown automaton which can be automatically generated by an LL(1) parser-generator.

↑ Phew parse that !

For more details on top down parsing and further exercises see

[Cooper - Chapter 3],
[Appel – Chapter 3] or
[Aho et al – Chapter 4]

ANTLR4 Parser Generator

Introduction

Naranker Dulay
n.dulay@imperial.ac.uk

https://www.doc.ic.ac.uk/~nd/compilers

Introduction

ANTLR (Another Tool for Language Recognition) is one most popular production quality parser generator tools. It's been developed and maintained by Terence Parr at the University of San Francisco for over 2 decades.

- Generates **recursive-descent parsers**. Incorporates **lexical analysis.**
- Based on 'adaptive' LL(*) grammars. Handles lots of grammatical issues cleanly without need for rewriting grammar, e.g. **(direct) left recursion**, **operator priorities**, **right-associative operators** like ^ and =,
- Will **lookahead** as many tokens as required.
- Produces **parse trees, not ASTs** i.e. terminal symbols are kept.
- Supports both the ***visitor*** **pattern** and the ***listener*** **pattern**.
- Written in Java, but can generate parsers in **Java**, **Python**, **Javascript**, **C#**
- Used in Twitter search query language, Groovy, Hibernate, Cassandra, Processing, Jython, many many more

Installation - Java

The following *may* help to setup ANTLR4 on your computer. Notes are based on Mac Installation. Linux should be similar.

- Install Java from Oracle.
- Download ANTLR4 (`antlr-4.5.3-complete.jar`) from `antlr.org` and place somewhere convenient e.g. `~/lib/`

  Add to Java's CLASSPATH. For example, add
  `export CLASSPATH=‘‘.:~/lib/antlr-4.5.3-complete.jar:$CLASSPATH’’`
  to `~/.bash_profile`
- On Mac's it useful to have `Xcode` installed. Download from the App Store

Installation – Python3

If you want to use Python3 as a target:

- Install Python3 if not already installed.. Mac's have Python2 pre-installed. You can download Python 3.5.2 from python.org.

- Download and install the latest ANTLR4 Python3 runtime from
  `https://pypi.python.org/pypi/antlr4-python3-runtime`

  Uncompress `antlr4-python3-runtime-4.5.3.tar.gz` and move the sub-directory `src/antlr4` to somewhere convenient, e.g `~/lib/python` which would have the sub-directory `~/lib/python/antlr4`

  Add to Python's package path. For example, add
  ```
  export PYTHONPATH='.:~/lib/python:$PYTHONPATH'
  ```
  to `~/.bash_profile`

ANTLR4 Syntax

ANTLR4 has a very natural, easy to use syntax:

- **Lexical rules** start with an **uppercase letter**, e.g. **Identifier**, **INT**
- **Parser rules** start with a lowercase letter, e.g. **expr**, **statement**
- The **alternatives** of a rule can be labelled with ***#label*** at the end of the alternative (see later for use).
- Lexical rules can be split into a lexical file (starting with **lexer grammar**) and imported into a parser rules file (starting with **grammar**) using an **import** directive.
- -> **skip** is special ANTLR syntax to discard matched input string, e.g. whitespace
- **Tokens** (terminals) map to class names ending in Node.
- **Rules** (non-terminals) map to classes ending in **Context**. Context classes include methods for each token and rule occurring on the RHS.

Example 1

Section 1.2

Hello.g4

```
grammar Hello;
prog  : 'hello' ID ;      // Matches hello followed by an identifier
ID    : [a-z]+ ;          // Identifier
EOL   : '\r'? '\n' ;      // End-Of-Line
WS    : [ \t]+ -> skip ;  // skip spaces and tabs
```

makefile

```
JAR    = /Users/nd/lib/antlr-4.5.3-complete.jar
ANTLR  = java -jar $(JAR) -no-listener

HelloParser.class: Hello.g4
        $(ANTLR) Hello.g4
        javac Hello*.java
```

ls

```
Hello.g4                      HelloParser$ProgContext.class
Hello.tokens                  HelloParser.class
HelloLexer.class              HelloParser.java
HelloLexer.java               makefile
HelloLexer.tokens
```

Example 1 - Python

Hello.g4

```
grammar Hello;
prog  : 'hello' ID ;       // Matches hello followed by an identifier
ID    : [a-z]+ ;           // Identifier
EOL   : ‘\r’? ‘\n’ ;       // End-Of-Line
WS    : [ \t]+ -> skip ;   // skip spaces and tabs
```

makefile

```
ANTLR=java -cp "/Users/nd/lib/antlr-4.5.3-complete.jar" org.antlr.v4.Tool \
           -Dlanguage=Python3 -no-listener

HelloParser.py:  Hello.g4
         $(ANTLR) Hello.g4
```

ls

```
Hello.g4            HelloLexer.py       HelloParser.py
Hello.tokens        HelloLexer.tokens   makefile
```

ANTLR command options

makefile

```
JAR     = /Users/nd/lib/antlr-4.5.3-complete.jar
ANTLR   = java -jar $(JAR) -no-listener
TESTRIG = java org.antlr.v4.runtime.misc.TestRig

HelloParser.class: Hello.g4
        $(ANTLR) Hello.g4
        javac Hello*.java

tree:   HelloParser.class                  # print parse tree in LISP form
        $(TESTRIG) Hello prog -tree

tokens: HelloParser.class                  # print tokens as structure data
        $(TESTRIG) Hello prog -tokens

gui:    HelloParser.class
        $(TESTRIG) Hello prog -gui         # display parse tree in window

ps:     HelloParser.class                  # generate postscript file
         $(TESTRIG) Hello prog -ps Hello.ps
```

Example 2 - Expressions Section 4.1

- Let's consider a simple expression language.
- For clarity we'll split the specification of the lexical tokens from the grammar.

ElvisTokens.g4

```
lexer grammar ElvisTokens;
ID   : [a-zA-Z]+ ;          // Identifier
INT  : [0-9]+ ;             // Integer
EOL  : '\r'? '\n' ;         // End-Of-Line
WS   : [ \t]+ -> skip ;     // skip spaces and tabs
MUL  : '*' ;
DIV  : '/' ;
ADD  : '+' ;
SUB  : '-' ;
```

Example 2 contd

- Let's consider a simple expression language.

Elvis.g4

```
grammar Elvis;
import ElvisTokens;

prog : cmd+ ;

cmd  : expr EOL
     | ID ‘=’ expr EOL
     | EOL
     ;

expr : expr (‘*’|‘/’) expr
     | expr (‘+’|‘-’) expr
     | INT
     | ID
     | ‘(‘ expr ‘)‘
     ;
```

Left recursive and starts more than 1 alternative!!

Earlier alternative has higher precedence

Assumes operators are left associative unless suffixed by <assoc=right>

RocknRoll.java

```java
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class RocknRoll {
  public static void main(String[] args) throws Exception {
    // create a CharStream that reads from standard input
    ANTLRInputStream input = new ANTLRInputStream(System.in);
    // create a lexer that reads from the input CharStream
    ElvisLexer lexer = new ElvisLexer(input);

    // create a buffer of tokens read from the lexer
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // create a parser that reads from the tokens buffer
    ElvisParser parser = new ElvisParser(tokens);

    // begin parsing at prog rule
    ParseTree tree = parser.prog();

    // print a LISP-style parse tree
    System.out.println(tree.toStringTree(parser));
   }
}
```

Functional RocknRoll

- Functional style

RocknRoll.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class RocknRoll {
  public static void main(String[] args) throws Exception {

     // functional style
     ElvisParser parser = new ElvisParser(
                                new CommonTokenStream(
                                  new ElvisLexer(
                                    new ANTLRInputStream(System.in))));

     // print a LISP-style parse tree
     System.out.println(tree.toStringTree(parser));
 }
}
```

Python RocknRoll

- In Python

RocknRoll.py

```
from antlr4 import *
from ElvisLexer import ElvisLexer
from ElvisParser import ElvisParser

def main(argv):
  input  = FileStream(argv[1])
  lexer  = ElvisLexer(input)
  stream = CommonTokenStream(lexer)
  parser = ElvisParser(stream)
  tree   = parser.prog()
  print(tree.toStringTree(recog=parser))

if __name__ == '__main__':
  import sys
  main(sys.argv)
```

Example 3 Calculator

Section 4.2

- Lets write an interpreter to calculate our expressions using a visitor

Elvis.g4

Label alternatives like this

```
grammar Elvis;
import ElvisTokens;

prog : cmd+ ;

cmd   : expr EOL                                  # PrintExpr
      | ID '=' expr EOL                           # Assign
      | EOL                                       # Blank
      ;

expr  : expr op=('*'|'/') expr                    # MulDiv
      | expr op=('+'|'-') expr                    # AddSub
      | INT                                       # Int
      | ID                                        # Id
      | '(' expr ')'                              # Parens
      ;
```

Visitor Interface

- Add **–visitor** flag to ANTLR4 to produce visitor support code.
- Context classes correspond to rules or labelled alternatives

ElvisVisitor.java

```
public interface ElvisVisitor<T> {

  T visitProg(ElvisParser.ProgContext ctx);          # rule Id

  T visitPrintExpr(ElvisParser.PrintExprContext ctx);   # label PrintExpr

  T visitAssign(ElvisParser.AssignContext ctx);      # label Assign

  T visitBlank(ElvisParser.BlankContext ctx);        # label Blank

  T visitMulDiv(ElvisParser.MulDivContext ctx);      # label MulDiv

  ...
}
```

Java Interpreter 1

- Our Interpreter is a subclass of ElvisBaseVisitor.

Interpreter.java

```
import java.util.HashMap;
import java.util.Map;

public class Interpreter extends ElvisBaseVisitor<Integer> {

  /** "memory" for our interpreter; variable/value pairs go here */
  Map<String, Integer> memory = new HashMap<String, Integer>();

/** expr EOL */
@Override
public Integer visitPrintExpr(ElvisParser.PrintExprContext ctx) {
  Integer value = visit(ctx.expr());          // evaluate expr
  System.out.println(value);                  // print the result
  return 0;                                   // return dummy value
}
```

Java Interpreter 2

Interpreter.java

```
/** ID */
@Override
public Integer visitId(ElvisParser.IdContext ctx) {
  String id = ctx.ID().getText();      // return value of ID else 0
  if (memory.containsKey(id)) return memory.get(id) else return 0;
}

/** INT */
@Override
public Integer visitInt(ElvisParser.IntContext ctx) {
  return Integer.valueOf(ctx.INT().getText());
}

/** ID '=' expr EOL */
@Override
public Integer visitAssign(ElvisParser.AssignContext ctx) {
  String id = ctx.ID().getText();       // Get ID
  int value = visit(ctx.expr());        // Get value of expr
  memory.put(id, value);                // Store in ID
  return value;
}
```

Java Interpreter 3

Interpreter.java

```
/** expr op=('*'|'/') expr */
@Override
public Integer visitMulDiv(ElvisParser.MulDivContext ctx) {
  int left = visit(ctx.expr(0));                // get value of left expr
  int right =  visit(ctx.expr(1));              // get value of right expr
  if (ctx.op.getType() == ElvisParser.MUL)
    return left * right;
  else return left / right;
}


/** expr op=('+'|'-') expr */
@Override
public Integer visitAddSub(ElvisParser.AddSubContext ctx) {
  int left  = visit(ctx.expr(0));               // get value of left expr
  int right = visit(ctx.expr(1));               // get value of right expr
  if (ctx.op.getType() == ElvisParser.ADD)
        return left + right;
  else return left - right;
}
```

Java Interpreter 4

Interpreter.java

```java
/** '(' expr ')' */
@Override
public Integer visitParens(ElvisParser.ParensContext ctx) {
  return visit(ctx.expr());                    // return value of expr
}
```

RocknRoll.java

```
In RocknRoll.java replace
        System.out.println(parser.prog().toStringTree(parser));
with
        Interpreter interpreter = new Interpreter();
        interpreter.visit(parser.prog);
```

Python Interpreter 1

Interpreter.py

```
from ElvisVisitor import ElvisVisitor
from ElvisParser  import ElvisParser

class Interpreter(ElvisVisitor):
  memory = {}

  def visitPrintExpr(self, ctx):
    value = self.visit(ctx.expr())
    print(value)
    return 0

  def visitId(self, ctx):
    id = ctx.ID().getText()
    if id in self.memory:
      return self.memory[id]
    return 0

  def visitInt(self, ctx):
    return int(ctx.INT().getText())
```

Python Interpreter 2

Interpreter.java

```
def visitAssign(self, ctx):
  id = ctx.ID().getText()
  value = self.visit(ctx.expr())
  self.memory[id] = value
  return value

def visitMulDiv(self, ctx):
  left = self.visit(ctx.expr(0))
  right = self.visit(ctx.expr(1))
  if ctx.op.type == ElvisParser.MUL:
    return left * right
  return left / right

def visitAddSub(self, ctx):
  left = self.visit(ctx.expr(0))
  right = self.visit(ctx.expr(1))
  if ctx.op.type == ElvisParser.ADD:
    return left + right
  return left - right
```

Python Interpreter 3

Interpreter.py

```
def visitParens(self, ctx):
   return self.visit(ctx.expr())
```

RocknRoll.py

```
from antlr4 import *
from ElvisLexer  import ElvisLexer
from ElvisParser import ElvisParser
from Interpreter import Interpreter

def main(argv):
  parser = ElvisParser(CommonTokenStream(ElvisLexer(FileStream(argv[1]))));
  tree = parser.prog ()
  interpreter = Interpreter()
  interpreter.visit(tree)      # or interpreter.visitProg(tree)

if __name__ == '__main__':
  import sys
  main(sys.argv)
```

More Information

- The Definitive ANTLR4 Reference, Terrence Parr. Pragmatic Bookshelf, 2012. You can download e-book version via Lab.
- Official Website: **www.antlr.org**