

# Compiler Construction

D7011E

## Lecture 6: Parser generators & abstract syntax

Contains material generously provided by Mark P. Jones



**COMPUTER SCIENCE  
AND ELECTRICAL ENGINEERING**  
LULEÅ UNIVERSITY OF TECHNOLOGY

# Parser Generators

- ◆ Yacc (Yet Another Compiler Compiler) – the classic bottom-up shift-reduce LALR(1) parser generator targeting C.
- ◆ Bison – a GNU variant of yacc.
- ◆ JavaCC – generates a yacc-like parser in Java.
- ◆ Happy – generates a yacc-like parser in Haskell.
- ◆ ...
- ◆ Coco/R – language-generic LL(k) parser generator.
- ◆ ANTLR – also language-generic, uses LL(\*).

# The Ins and Outs of ANTLR pt. 2

- ◆ If ANTLR is given a parser grammar in file `MyParser.g`, its output is a file `MyParser.java` defining a class `MyParser` that extends a library class `Parser`.

- ◆ The important methods of class `MyParser` are

```
void rule1();
```

```
...
```

```
void rulen();
```

if `MyParser.g` defines non-terminals `rule1` to `rulen`.

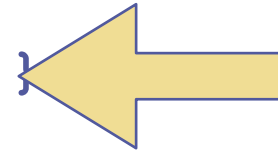
# ANTLR parser grammar format:

In file MyParser.g:

```
parser grammar MyParser;
```

```
... optional declarations ...
```

```
options { tokenVocab = MyLexer; }
```



```
... context-free grammar rules ...
```

# The Ins and Outs of ANTLR pt. 3

- ◆ If ANTLR is given a combined grammar in file `Test.g`, its output is:
  - file `TestParser.java` defining class `TestParser` that extends library class `Parser`
  - file `TestLexer.java` defining class `TestLexer` that extends library class `Lexer`
- ◆ The contents of classes `TestParser` and `TestLexer` is the same as if stand-alone grammars had been used.

# ANTLR combined grammar format:

In file Test.g:

```
grammar Test;
```

```
... optional declarations ...
```

```
... context-free grammar rules ...
```

```
... lexical rules (same as in lexer grammars) ...
```

# Rule Syntax

- ◆ An example of the schematic grammar rules we have used so far:

$A \rightarrow (E)$

$A \rightarrow n$

- ◆ The same rules expressed in ANTLR:

```
atom : '(' expr ')'
      | INT
      ;
```

- ◆ Differences: lower-case non-terminals, upper-case terminals, quoted terminal strings, colon and semicolon, explicit alternatives, no epsilons, ...

# The Quick Calculator, revisited:

- ◆ In the first lecture, we looked at the quick calculator program, comprising:
  - A lexical analyzer;
  - A recursive descent parser;
  - A representation for the abstract syntax of expressions, including code for evaluation and code generation;
  - A top-level driver.
- ◆ A few weeks later, what would we change?

# The Quick Calculator in ANTLR

## ◆ In Quick.g:

```
grammar Quick;
```

```
program    : expr program1
```

```
;
```

```
program1  : ';' expr program1
```

```
|
```

```
;
```

```
expr      : term expr1
```

```
;
```

```
expr1     : '+' term expr1
```

```
|
```

```
'-' term expr1
```

```
|
```

```
;
```

# The Quick Calculator in ANTLR

## ◆ In Quick.g:

```
term      : atom term1
          ;
term1     : '*' atom term1
          | '/' atom term1
          ;
atom      : INT
          | '(' expr ')'
          ;
INT       : '0'..'9'+
          ;
WS        : ( ' ' | '\n' | '\t' )+    { skip(); }
          ;
```

# Invoking the parser

## ◆ The main compiler driver:

```
public static void main(String[] args) throws Exception {  
    ANTLRInputStream input = new ANTLRInputStream(System.in);  
  
    QuickLexer lexer      = new QuickLexer(input);  
  
    CommonTokenStream tokens  
                                = new CommonTokenStream(lexer);  
  
    QuickParser parser     = new QuickParser(tokens);  
  
    parser.program();  
  
}
```

# The result of parsing

- ◆ Calling `parser.program()` will
  - silently succeed (`void` result) if the input text is syntactically correct (belongs to language Quick)
  - Throw an exception (`org.antlr.runtime.RecognitionException`) if the input text contains a syntactic error (does not belong to language Quick)
- ◆ Recalling `QuickCalc.java`, what we really want is
  - a generated abstract syntax tree corresponding to the input if it is syntactically correct
  - an exception otherwise

# Getting Some Action:

- ◆ We can annotate each production in the grammar with an action, containing code to execute when the production is reduced:

```
nonterminal      : alt1 { Java action 1 }  
                  |  
                  | altn { Java action n }  
                  ;
```

Some actions do  
things ...

Some actions pass  
things on ...

Some actions  
construct things ...

# Getting Some Action:

- ◆ Data associated with a grammar rule are commonly referred to as semantic values.
- ◆ To make rules produce semantic values we need to change the default void type of rule methods:

```
nonterminal    returns [ Expr tree ]  
  
: alt1  { $tree = Java expression 1 }  
|  
| altn  { $tree = Java expression n }  
;
```

# Getting Some Action:

- ◆ Rules that return trees and refer to subtrees (using two different notations):

```
atom  returns [ Expr tree ]
      : INT      { $tree = new IntExpr(str2int($INT.text)); }
      | '(' expr ')' { $tree = $expr.tree; }
      ;
```

```
expr  returns [ Expr tree ]
      : e1=atom '+' e2=atom
          { $tree = new BinExpr('+', $e1.tree, $e2.tree); }
      ;
```

# Getting Some Action:

- ◆ Rules do not necessarily have to return abstract syntax trees:

```
atom  returns [ int val ]
      : INT          { $val = str2int($INT.text); }
      | '(' expr ')' { $val = $expr.val; }
      ;
```

```
expr  returns [ int val ]
      : e1=atom '+' e2=atom
          { $val = $e1.val + $e2.val; }
      ;
```

# Getting Some Action:

- ◆ Rules do not even have to return data:

```
program1      : ';' expr program1
               { System.out.println($expr.val); }
               | { /* empty! */ }
               ;
```

- ◆ I.e., side-effects in general are allowed (but might of course sometimes be tricky to understand)

# Abstract syntax

- ◆ A particular semantic value that can be constructed is a representation of the parsed input (i.e., the parse tree)
- ◆ This is typically a representation that is much simpler than concrete syntax:
  - No parenthesized expressions
  - No disambiguating tricks like `expr/term/factor/atom`
  - No redundant keywords or separators, just simple node tags
  - Explicit lists instead of recursion
  - etc...
- ◆ The structure of these parse trees can also be described by a grammar – the abstract program syntax

# Abstract syntax in Java

◆ For QuickCalc we could have defined:

Expr	→	BinExpr Expr op Expr
Expr	→	IntExpr int

◆ Note: no need to remember parentheses anymore!

◆ Implemented in Java as:

- An abstract class for each nonterminal (**Expr**)
- Methods that reflect what we want to do with the abstract representation (**eval**, **codegen** (**,** **print**, **typecheck**, ...))
- A concrete subclass for each production for an abstract syntax nonterminal (**BinExpr**, **IntExpr**)
- For each concrete class: fields for each right-hand side component (and corresponding constructor parameters)

# Abstract syntax in Java

```
abstract static class Expr {  
    abstract void compile();  
    abstract int  eval();  
}
```

Limitation: what we want  
to do must be defined  
in advance!

Solution: Visitors!  
(more next lecture)

```
static class BinExpr extends Expr {  
    private char op;  
    private Expr left;  
    private Expr right;  
    BinExpr(char op, Expr left, Expr right);  
    ...  
}
```

```
static class IntExpr extends Expr {  
    private int value;  
    IntExpr(int value);  
    ...  
}
```

# Rule Syntax

- ◆ So far, we have used the pure BNF (Backus-Naur form) notation for context-free grammars to express repetition using recursion and empty alternatives:

```
program      : expr program1 ;
program1     : ';' expr program1
              |
              ;
```

- ◆ The EBNF (Extended Backus-Naur form) of ANTLR provides a convenient alternative:

```
program      : expr (';' expr)* ;
```

- ◆ Note: a regular expression operator!

# Recursion vs. Repetition

- ◆ Repetition `regexp*` (also known as the Kleene star) is just a special form of recursion pattern:

```
newname    : regexp newname
           |
           ;
```

- ◆ That is, right-recursion only, plus an empty alternative
- ◆ The operator `regexp+` is defined similarly:

```
newname    : regexp newname
           | regexp
           ;
```

- ◆ In fact, regular expression = context-free grammars restricted to right-recursion only!

# Our calculator, using repetition

## ◆ In Quick.g:

grammar Quick;

program : expr ( ';' expr )\* ;

expr : term ( ('+'|'-') term )\* ;

term : atom ( ('\*'|'/') atom )\* ;

atom : INT  
| '(' expr ')';

INT : '0'..'9'+ ;

WS : ( ' ' | '\n' | '\t' )+ { skip(); }



# Repetition and actions

- ◆ A built-in feature of ANTLR: accumulation of semantic values using an **ArrayList**:

```
program    returns [ArrayList result]
          : es+=expr ( ';' es+=expr )*
          { $result = $es; }
```

- ◆ I.e., `es` becomes a local variable of type **ArrayList**, initialized to the empty list and appended with each semantic value referred to using the **+=** notation.

# LR(1) > LL(k)

- ◆ A language that is in LR(1) (actually LALR(1)) but not in LL(k) for any finite k:

```
expr : aa '!'  
      | bb '?'
```

How choose production in advance?

```
aa : (INT '+' ) *  
;
```

Both A and B might be arbitrarily long...

```
bb : (INT SYM) *  
;
```

But A and B can also differ at arbitrarily many places!

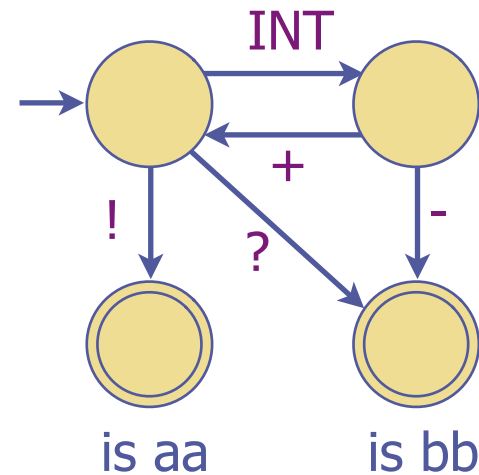
```
SYM : '+' | '-' ;
```

# LL(\*) > LL(k)

- ◆ However, the same language is in LL(\*) (despite not being in LL(k) for any finite k!)

```
expr : aa '!'
      | bb '?'
;
aa   : (INT '+' ) *
;
bb   : (INT SYM) *
;
SYM  : '+' | '-' ;
```

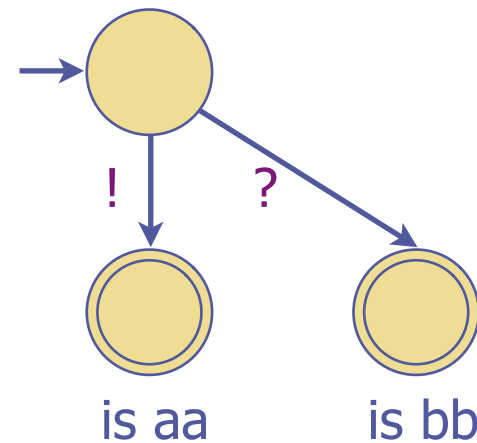
Let a custom-made DFA make the choice!



# The DFA of an LL(1) rule

- ◆ A DFA is actually implicitly used even for simple predictions:

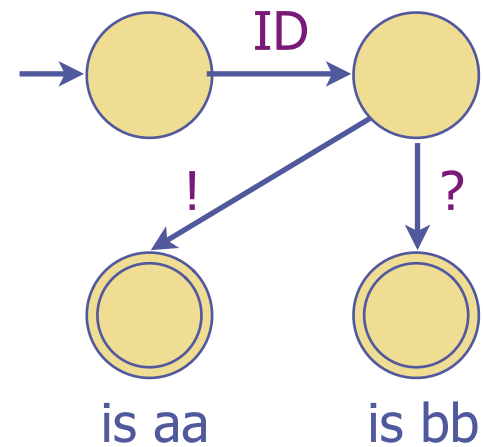
```
expr : '!' aa  
      | '?' bb  
      ;
```



# The DFA of an LL(2) rule

◆ ... and for slightly more involved predictions:

```
expr : ID '!' aa  
      | ID '?' bb  
      ;
```



# The power of ANTLR predictions

- ◆ Even recursion in one of the predictions is ok:

```
expr : aa '!'
      | bb '?'
      ;
aa   : '(' aa ')'
      | INT
      ;
bb   : '(' INT ')'
      ;
```

# The limits of ANTLR predictions

- ◆ However, recursion in two or more alternatives forces ANTLR to complain:

```
expr : aa '!'  
      | aa '?'  
;  
aa   : '(' aa ')'  
      | INT  
;
```

- ◆ This restriction is an overapproximation, but exists to stay on the right side of non-termination.

# Circumventing prediction limitations

- ◆ Still, ANTLR can be told to do the "obvious" test – try each alternative and pick the first success.

```
expr options { backtrack = true; }
    : aa '!'
    | aa '?'
    ;
aa   : '(' aa ')'
    | INT
    ;
```

- ◆ Backtracking results in exponential complexity, though, so it should be used with care!

# True ambiguity: the Dangling Else

- ◆ A famous problem in the design of programming language parsers:

```
stmt      :  IF expr THEN stmt ELSE stmt
          |  IF expr THEN stmt
          |  ...
          ;
```

- ◆ This grammar is ambiguous!
- ◆ Causes a prediction failure in ANTLR
- ◆ Causes a shift/reduce conflict in yacc

# Eliminating the Dangling Else:

- ◆ Rewrite the grammar to eliminate the ambiguity, without changing the language:

```
stmt      : matched
           | unmatched
           ;
matched   : IF expr THEN matched ELSE matched
           | other
           ;
unmatched : IF expr THEN stmt
           | IF expr THEN matched ELSE unmatched
           ;
```

# Grammar Rewriting Woes:

- ◆ The declarative ideal:
  - You know your parser is correct, because it was generated directly from your grammar.
- ◆ The practical reality:
  - Your grammar might not be good enough for the parser generator.
- ◆ Changing the grammar:
  - May obfuscate and complicate it;
  - May introduce errors;
  - May increase the size of the parser;
  - May require duplicated actions.

# An example from eqn:

◆ The `eqn` package for typesetting mathematics uses the following notation for subscripts and superscripts:

■ `foo sub bar`  $\Rightarrow$  `foobar`

■ `foo sup bar`  $\Rightarrow$  `foobar`

◆ The corresponding grammar has conflicts:

```
exp      : exp SUB exp
         | exp SUP exp
```

...

◆ (These are resolvable in `yacc` using a precedence directive `%right` ).

# A reduce/reduce conflict:

- ◆ But now suppose that you want both a subscript and a superscript ...

- foo sub bar sup baz  $\Rightarrow$  foo<sub>bar</sub><sup>baz</sup>

- foo sup baz sub bar  $\Rightarrow$  foo<sup>baz</sup><sub>bar</sub>

But we want:

foo<sup>baz</sup><sub>bar</sub>

- ◆ We can add a special case to the grammar:

exp : exp SUB exp SUP exp

      | exp SUB exp

      | exp SUP exp

...

- ◆ But now we get a reduce/reduce conflict in yacc

# Reflections on LALR conflicts:

- ◆ **Shift/reduce** conflicts occur when there is not enough information for the parser to decide what to do next. There may or may not be an ambiguity.
- ◆ **Reduce/reduce** conflicts are an indication of certain ambiguity, and are usually caused by serious problems in the grammar.
- ◆ Normally, you will have to rewrite the grammar.

# Left-factorization

- ◆ An alternative to backtracking in LL parsers is to left-factorize the grammar:

```
exp : term '('exp ')'      { call action }  
    | term '['exp ']'      { array action }  
    ...
```

- ◆ ... is rewritten to

```
exp : term tail  
    |  
    ...  
  
tail : '('exp ')'      { modified call action }  
      '['exp ']'      { modified array action }
```

# Left / Right Recursion:

- ◆ A production of the form  $A \rightarrow A w$  is said to be left recursive.
- ◆ A production of the form  $A \rightarrow w A$  is said to be right recursive.
- ◆ Many languages can be expressed using either left or right recursion:

```
prog : prog ';' expr  
      | expr  
      ;
```

```
prog : expr ';' prog  
      | expr  
      ;
```

# Comparing left and right:

◆ Recall that shift/reduce parsers produce a rightmost derivation (in reverse).

◆ With a left recursive grammar:

$e\_;e;e;e \rightarrow p;e\_;e;e \rightarrow p;e\_;e \rightarrow p;e\_ \rightarrow p$

◆ With a right recursive grammar:

$e;e;e;e\_ \rightarrow e;e;e;p\_ \rightarrow e;e;p\_ \rightarrow e;p\_ \rightarrow p$

◆ A right recursive production delays reduction by pushing everything on the stack.

# Bottom-up favors left recursion:

- ◆ Both are acceptable grammars;
- ◆ Both result in parsers that take a list of expressions and display the corresponding results.

BUT:

- ◆ The right recursive version uses more stack space;
- ◆ Doesn't give a result until all expressions have been entered;
- ◆ Displays the results in reverse order!

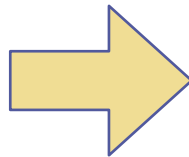
# Top-down chokes on left recursion:

- ◆ Left-recursive grammars may lead to non-terminating parsers;
- ◆ ANTLR blindly refuses to accept left-recursive grammars.

BUT:

- ◆ Every left-recursive grammar can be converted into a non-left-recursive one!

```
foo : foo bar1  
    | bar2
```



```
foo   : bar2 foo1  
foo1  : bar1 foo1  
      |
```

- ◆ Mutual recursion complicates the patterns, though.

# Summary:

- ◆ LL prediction problems will appear, and so will LALR conflicts
- ◆ A conflict isn't necessarily an error ... but you should check conflict reports very carefully.
- ◆ Rewriting grammars can help to eliminate problems, but can also introduce new problems.
- ◆ Tools like ANTLR can be useful in the early stages of language development as a tool for grammar design/debugging, not just parser construction.
- ◆ Next time: Introducing semantic analysis ...