Introduction to Lexing and
Parsing Techniques*

Nick Papanikolaou

nikos@dcs.warwick.ac.uk
http://www.warwick.ac.uk/go/nikos

Lecture 2: **Syntactic Analysis - Bottom–Up Parsing**

1 / 35

[a] As part of module CS245: Automata and Formal Languages.

Introduction 2 / 35

Overview

Last lecture we considered various properties of grammars, including the issue of *uniqueness of derivations*, *self-embedding* and *ambiguity*.

Also, we looked at the **limitations** of CFGs and the concept of an **attribute grammar**.

We discussed the process of **parsing** in general, and we looked at the construction of a **recursive-descent parser** for a subset of Pascal types. In particular, we built a predictive parser, which did not need to backtrack.

3 / 35

Overview

The techniques we considered all fall under the heading of "top–down parsing". In this lecture we will:

- review **top–down parsing** briefly;
- discuss the **advantages** and **pitfalls** of that technique;
- revisit `javacc`, which generates top–down parsers;
- look at **bottom–up** parsing.

4 / 35

Review of Top–down Parsing

Top–down parsing tries to construct a parse tree for the **leftmost derivation** of an expression. It starts at the **root** of the parse tree and works downward.

As we saw in the last lecture, top–down parsing consists of **two** steps:

- At node n (non–terminal A), select one of the productions for A and construct children for the symbols on the RHS.
- Find the next non–terminal with no children. If lookahead symbol matches current terminal symbol, proceed to next input character.

6 / 35

Knuth's Notation for Parsers

Top–down parsers and the unambiguous grammars they are capable of handling with k symbols of lookahead are known as **LL**(**k**). Similarly, bottom–up parsers are referred to as **LR**(**k**).

The parser we built in the last lecture was **LL**(**1**).

LL(**k**) left-to-right, **leftmost** derivation

LR(**k**) left-to-right, **rightmost** derivation

SLR() "simple LR" – better than **LR**(**0**)

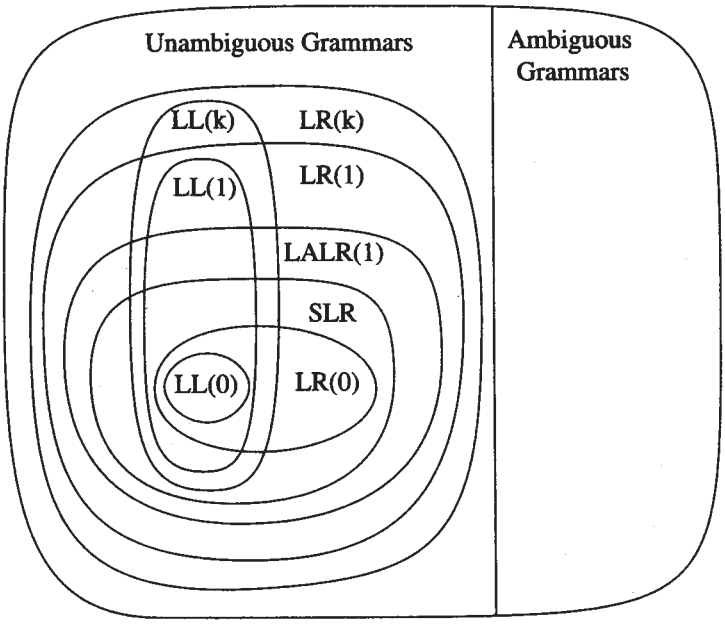
LALR(**k**) "lookahead LR"

The last two represent techniques with improvements to the way in which the parse table is constructed.

Remember the **hierarchy** of unambiguous grammars.

7 / 35

Hierarchy of Grammar Classes



8 / 35

Advantages of LL(1) Parsing

Recursive–descent parsing applies to **LL(1)** grammars. Its key benefits are:

- The technique is appealing because it seems **natural**; it is used extensively in teaching.
- It is **easy** to implement and to have **confidence** in its correctness;
- The functions written can include **actions** for (mainly type) **checking**;
- Actions may include **synthesis**, esp. storage allocation and code generation.

9 / 35

Disadvantages of LL(1) Parsing

Problems with recursive descent include:

- the **inefficiency** of recursive function calls;
- the need for **grammar transformations** (elimination of *left recursion* and *factorization*);
- very **large** parsers are often produced using this technique;
- the tendency for actions which are part of different phases of compilation to appear in the same functions in code.

Essential point: the effective use of the recursive descent technique requires a **grammar transformer** capable of transforming a grammar to **LL(1)** form. This is theoretically impossible for *every* possible input.

`javacc` Revisited

`javacc` is an **LL(k)** parser generator. In the first lecture, we only used its lexing features[a].

Have a look at:

`https://javacc.dev.java.net/` `http://www.engr.mun.ca/~theo/JavaCC-FAQ/`

Let's demonstrate `javacc` by building a recognizer for **matching parentheses**.

[a] In `javacc`–speak, a lexer is known as a **token manager**.

A javacc parser for Matching Parentheses

```
// javacc-4.0/examples/SimpleExamples/Simple3.jj
// Part 1/3

PARSER_BEGIN(Simple3)
public class Simple3 {

  public static void main(String args[])
                   throws ParseException
  {
    Simple3 parser = new Simple3(System.in);
    parser.Input();
  }
}
PARSER_END(Simple3)
```

12 / 35

A javacc parser for Matching Parentheses

```
// Part 2/3
SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN :
{
  <LBRACE: "{">
| <RBRACE: "}">
}
```

13 / 35

A `javacc` parser for Matching Parentheses

```
// Part 3/3
void Input() :
{ int count; }
{
  count=MatchedBraces() <EOF>
  { System.out.println("The level of
    nesting is " + count); }
}

int MatchedBraces() :
{ int nested_count=0; }
{
<LBRACE> [ nested_count=MatchedBraces() ] <RBRACE>
  { return ++nested_count; }
}
```

14 / 35

A `javacc` parser for Matching Parentheses

To compile and run this parser:

```
$ javacc Simple3.jj
$ javac *.java
$ java Simple1
{x
    Lexical error at line 1, column 2.
    ...
{}
    (ok)
```

The current directory must be in your `CLASSPATH`.

Read and work through the `README` file for the `SimpleExamples`.

15 / 35

Shift–Reduce Parsing

Bottom–up parsing attempts to construct a parse tree beginning at the leaves and working towards the root.

We will consider a standard style of bottom–up parsing known as **shift–reduce** parsing. This is used in tools such as `yacc`.

- The idea is to "reduce" an input string w to the **start symbol** of the grammar.
- At each reduction step, a substring matching some RHS of a production is replaced by the symbol on the LHS.
- If the substring is chosen correctly at each step, a **rightmost derivation** is traced out in reverse.

17 / 35

Example of Bottom-Up Parse

We want to parse the sentence $\boxed{\texttt{abbcde}}$, belonging to the language generated by a grammar with productions:

$$\begin{aligned} S &::= \texttt{a}A\texttt{B}\texttt{e} \\ A &::= A\texttt{bc} \\ &\mid \texttt{b} \\ B &::= \texttt{d} \end{aligned}$$

$$\boxed{\texttt{abbcde}} \rightarrow \boxed{\texttt{a}A\texttt{bcde}} \rightarrow \boxed{\texttt{a}A\texttt{de}} \rightarrow \boxed{\texttt{a}A\texttt{B}\texttt{e}} \rightarrow \boxed{S}$$

We have computed, in reverse, the rightmost derivation

$$S \overset{\text{rm}}{\Rightarrow} \texttt{a}A\texttt{B}\texttt{e} \overset{\text{rm}}{\Rightarrow} \texttt{a}A\texttt{de} \overset{\text{rm}}{\Rightarrow} \texttt{a}A\texttt{bcde} \overset{\text{rm}}{\Rightarrow} \texttt{abbcde}$$

18 / 35

Handles

The green substrings in the last slide are known as **handles**.

Informally, a **handle** of a string is a substring that matches the RHS of a production, and whose reduction to the non–terminal on the LHS represents one step along the reverse of a rightmost derivation.

Example 1. *If*

$$S \overset{rm^*}{\Rightarrow} \alpha A w \overset{rm}{\Rightarrow} \alpha\beta w$$

then $A ::= \beta$ *in the second position is a handle of* $\alpha\beta w$.

More on Handles

Definition 1. *A **right sentential form** is one which occurs in some step of a rightmost derivation. A **sentence** is a special right sentential form that consists entirely of terminals.*

Definition 2. *A **handle** of a right sentential form* γ *is a production* $A ::= \beta$ *and a position of* γ *where the string* β *may be found and replaced by* A *to produce the previous right sentential form in a rightmost derivation of* γ.

Bottom–up parsing essentially consists in finding handles repeatedly, starting from the sentence we're working on and proceeding until the start symbol of the grammar is reached.

Implementing a Shift–Reduce Parser

A S/R parser has a **stack** to hold grammar symbols, and an **input buffer** to hold the string w to be parsed.

- Initially, the stack is empty and the string w is in the input.
- The parser operates by **shifting zero or more input symbols** onto the stack until a handle β is on top of the stack.
- Then the parser **reduces** β to the LHS of the appropriate production.
- This process is repeated until the parser finds an error or the stack contains the start symbol and the input has been exhausted.

Implementing a Shift–Reduce Parser

A S/R parser has four different types of actions:

shift the next symbol is shifted onto the stack.

reduce the parser replaces the handle on the stack with a non–terminal.

accept the parser announces successful completion of parsing.

error the parser discovers and reports an error.

Example of S/R Parse

Consider the sentence

$$\mathbf{id_1} + \mathbf{id_2} \times \mathbf{id_3}$$

This sentence belongs to a language whose grammar consists of the productions:

$$\begin{aligned} \mathtt{E} &::= \mathtt{E} + \mathtt{E} \\ \mathtt{E} &::= \mathtt{E} \times \mathtt{E} \\ \mathtt{E} &::= \text{'('} \; \mathtt{E} \; \text{')'} \\ \mathtt{E} &::= \mathbf{id} \end{aligned}$$

Example of S/R Parse

Stack	Input	Action
\$	$\mathbf{id_1} + \mathbf{id_2} \times \mathbf{id_3}\$$	shift
\$ $\mathbf{id_1}$	$+\mathbf{id_2} \times \mathbf{id_3}\$$	reduce ($\mathtt{E} ::= \mathbf{id}$)
\$ E	$+\mathbf{id_2} \times \mathbf{id_3}\$$	shift
\$ E+	$\mathbf{id_2} \times \mathbf{id_3}\$$	shift
\$ E + $\mathbf{id_2}$	$\times\mathbf{id_3}\$$	reduce ($\mathtt{E} ::= \mathbf{id}$)
\$ E + E	$\times\mathbf{id_3}\$$	shift
\$ E + E×	$\mathbf{id_3}\$$	shift
\$ E + E × $\mathbf{id_3}$	\$	reduce ($\mathtt{E} ::= \mathbf{id}$)
\$ E + E × E	\$	reduce ($\mathtt{E} ::= \mathtt{E} \times \mathtt{E}$)
\$ E + E	\$	reduce ($\mathtt{E} ::= \mathtt{E} + \mathtt{E}$)
\$ E	\$	accept

S/R Conflicts

In the example parse, why wasn't $E + E$ reduced to E before shifting $\times$ onto the stack?

This is known as a **shift/reduce conflict** because the parser wouldn't know whether to shift or to reduce.
In order to resolve such conflict, we need to specify **operator precedence** and **associativity** information.

Thus, the grammar

$$E ::= E + E \mid E \times E \mid (E) \mid \textbf{id}$$

is ambiguous because it does not specify the precedence and associativity of $+$ and $\times$.

Parser generators can detect such conflicts and enforce one possibility instead of the other.

25 / 35

The `yacc` Parser Generator

Since we studied `lex` in the first lecture, we ought to consider its companion program `yacc` for the sake of completeness.

- `yacc` is a **LALR** parser generator and is an abbreviation for "Yet Another Compiler–Compiler."
- The command `yacc spec.y` generates a C program `y.tab.c` from the specification `spec.y`.
- The parser in `y.tab.c` must be compiled as follows: `gcc y.tab.c -ly`.
- *Note:* for `lex` output, similarly: `gcc main.c -ll`.

26 / 35

A Simple Calculator

```
// Part 1/2
%{
#include <ctype.h>
%}
%token DIGIT
%%
line   : expr '\n'       { printf("%d\n", $1); } ;
expr   : expr '+' term   { $$ = $1 + $3; }
       | term ;
term   : term '*' factor { $$ = $1 * $3; }
       | factor ;
factor : '(' expr ')'    { $$ = $2; }
       | digit ;
%%
```


A Simple Calculator

```
// Part 2/2
// The lexer:
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
// This last part could have been omitted,
// and lex used instead.
```

The parser generated by `yacc` from this input file calculates the value of simple arithmetic expressions.
It resolves the S/R conflict automatically.

Summary

In this lecture:

- we revised **top–down parsing,** and **LL**(**1**) parsing in particular;
- we considered the **pros** and **cons** of **LL**(**1**) parsing;
- we used `javacc` to make a parser for matching parentheses;
- we learned about bottom–up (**LR**) parsing;
- we implemented a **LALR** calculator using `yacc`.

30 / 35

Putting it all together

In this series of lectures, you learned about lexical and syntactic analysis, the main phases of any compiler.

You will have the opportunity to revisit this material in course **CS325: Compiler Design**.

By now, you should understand the connection between language and automata theory, and you know something about the different classes of grammars and how to implement recognizers for them.

31 / 35

How bad can it get?

```
1        INTEGERFUNCTIONA
2        PARAMETER(A=6,B=2)
3        IMPLICIT CHARACTER*(A-B)(A-B)
4        INTEGER FORMAT(10),IF(10),DO9E1
5 100    FORMAT(4H)=(3)
6 200    FORMAT(4 )=(3)
7        DO9E1=1
8        DO9E1=1,2
9            IF(X)=1
10           IF(X)H=1
11           IF(X)300,200
12 300       CONTINUE
13       END
     C   this is a comment
       $ FILE(1)
14       END
```

Example due to Dr. F.K. Zadeck of IBM Corp.

32 / 35

Reading

[1] Aho, Sethi, and Ullman, **Compilers: Principles, Techniques, and Tools**, Addison–Wesley, 1986.

[2] Tremblay and Sorenson, **The Theory and Practice of Compiler Writing**, McGraw–Hill, 1985.

[3] Appel, **Modern Compiler Implementation In Java**, 2nd ed., Cambridge, 2002.

[4] Hunter, **The Essence of Compilers**, Prentice–Hall, 1999.

Have a look also at the CS325 notes (`www.dcs.warwick.ac.uk/~sk/cs325/index.html`).

33 / 35

The End

I hope you have enjoyed this series of lectures.

Tip: Don't leave it all to the last minute... Study some of this now, before you accumulate a backlog! Good luck in the exam.

THE END

34 / 35