Introduction to Lexing and
Parsing Techniques*

Nick Papanikolaou

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

Lecture 2: **Syntactic Analysis - Top-Down Parsing**

1 / 44

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

Introduction

2 / 44

Introducing Parsing

As we have seen, **parsing** is part of the analysis phase of any compiler.

Parsing is the process of discovering the **structure** of a sentence in a particular language. In **natural language,** words change meaning depending on the context; natural language is thus **sensitive to context;** but that does not mean natural language is generated by a context–sensitive grammar!

The **syntax of programming languages** is expressed using CFGs. As we shall see, CFGs generate a **superset** of a programming language's actual syntax, and **constraints** must be imposed on the values of terminals in a CFG.

3 / 44

Outline

1. Derivations and Parse Trees

2. Recursion in Grammars

3. Ambiguous Grammars

4. Limitations of CFGs

5. Parsing in General

6. Top–Down Parsing

4 / 44

Derivations and Parse Trees

5 / 44

Review of Grammars and Languages

Remember, a **grammar** G is a quadruple (T, N, P, s) where:

- T is an alphabet of **terminal** symbols;
- N is an alphabet of **non–terminal** symbols;
- P is a set of **productions,** i.e. pairs (α, β) with $\alpha \in (T \cup N)^+$ and $\beta \in (T \cup N)^*$.
- s is known as the **start symbol** and $s \in N$.
- T and N have no symbols in common.

The **language generated by** G is written $L(G)$ and is a set consisting of all the strings, **sentential forms** or **sentences** that may be formed by applying the productions in G in all possible ways, starting from the start symbol s.

6 / 44

Example

Example 1. *The language* $\{x^n y^n \mid n > 0\}$ *is generated by the grammar*

$$G_1 = \{\{x, y\}, \{S\}, P, S\}$$

where $P = \{S ::= xSy, S ::= xy\}$

Does G_1 generate the string xxxxyyyy ?

To answer the question, we have to try to derive the string from the start symbol using the productions P. That is, we need to find a **derivation** $S \Rightarrow^+ \mathtt{xxxxyyyy}$.

$$S \Rightarrow xSy \Rightarrow xxSyy \Rightarrow xxxSyyy \Rightarrow xxxxyyyy$$

This derivation is **unique**.

7 / 44

More on Derivations

In general, derivations are **not** unique. **Regular grammars** always have a **unique derivation** for a given string because there is never more than one non–terminal on the right–hand side of a production.

The language $\{x^m y^n \mid m, n > 0\}$ is generated by a grammar G_2 with productions:

$$S ::= XY, \quad X ::= xX, \quad X ::= x, \quad Y ::= yY, \quad Y ::= y$$

There are several ways of generating the sentence xxxyy from this grammar, including:

$$S \Rightarrow XY \Rightarrow xXY \Rightarrow xxXY \Rightarrow xxxY \Rightarrow xxxyY \Rightarrow xxxyy \quad (1)$$
$$S \Rightarrow XY \Rightarrow XyY \Rightarrow Xyy \Rightarrow xXyy \Rightarrow xxXyy \Rightarrow xxxyy \quad (2)$$

8 / 44

Leftmost and Rightmost Derivations

- In (1), the leftmost non–terminal in the sentential form is replaced at each step. Hence (1) is known as the **leftmost derivation** of the sentence xxxyy.
- In (2), the rightmost non–terminal in the sentential form is replaced at each step. Hence (2) is known as the **rightmost derivation** of the sentence xxxyy.
- Other derivations for xxxyy are also possible from grammar G_2.

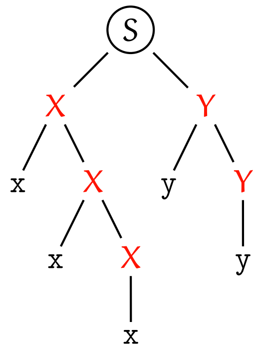
9 / 44

Parse Trees

A derivation may be equivalently expressed in 2D, in what is known as a **parse tree**. The leftmost derivation

$$S \Rightarrow XY \Rightarrow \mathtt{x}XY \Rightarrow \mathtt{xx}XY \Rightarrow \mathtt{xxx}Y \Rightarrow \mathtt{xxxy}Y \Rightarrow \mathtt{xxxyy}$$

is equivalent to the parse tree shown below.



Recursion in Grammars

Types of Recursion

The productions in a grammar usually contain recursion, of which there are three kinds:

$A ::= A\mathtt{b}$	(Left Recursion)
$B ::= \mathtt{c}B$	(Right Recursion)
$C ::= \mathtt{d}C\mathtt{f}$	(Middle Recursion or Self-embedding)

There is a useful recursion-related result which allows us to determine whether a grammar generates a regular language or a CFL.

Self-embedding Theorem. *If a grammar contains **no middle recursion** then the language it generates is regular.*

Note that, if there is no recursion at all in a grammar, then it is finite and therefore regular.

More on Self-embedding

Counterexample 1. *The language* $\{x^n y^n \mid n > 0\}$ *is generated by a grammar with productions* $S ::= xSy$ *and* $S ::= xy$*. Although this is a simple language it is **not** regular.*

This is relevant to compiler design because of the classic **bracket–matching problem**. Strings consisting of matching brackets are generated by a grammar with productions:

$$S ::= \text{'('}\ S\ \text{')'}$$
$$S ::= S\ S$$
$$S ::= \epsilon$$

The grammar is not regular, so you cannot build a lexer to recognize matching parentheses.

13 / 44

Ambiguous Grammars

14 / 44

Unambiguous Grammars

The following are equivalent statements regarding any grammar G:

- Each sentence generated by G has a unique leftmost derivation.
- Each sentence generated by G has a unique rightmost derivation.
- sentence generated by G has a unique parse tree.

A grammar with the above properties is **unambiguous**. Other grammars are ambiguous.

15 / 44

An Ambiguous Grammar

Example 2. *The grammar for producing sums of xs has productions*

$$S ::= S + S \qquad S ::= x$$

This grammar is ambiguous.

The string $x + x + x$ has two leftmost derivations:

$$\begin{aligned} S &\Rightarrow S + S \Rightarrow x + S \Rightarrow x + S + S \\ &\Rightarrow x + x + S \Rightarrow x + x + x \\ S &\Rightarrow S + S \Rightarrow S + S + S \Rightarrow x + S + S \\ &\Rightarrow x + x + S \Rightarrow x + x + x \end{aligned}$$

These derivations differ in a manner similar to $(x \cdot (x + x))$ vs. $((x \cdot x) + x)$.

16 / 44

Removing Ambiguity

The grammar mentioned previously can be made unambiguous by changing the productions to:

$$S ::= S + x \qquad S ::= x$$

This conversion is not always possible, and there is no general way of doing this. Determining whether a grammar is ambiguous is **undecidable**. Some well–known ambiguous grammars:

- grammars with one or more productions containing **left and right** recursion;
- the 'dangling `else`' grammar for imperative programming languages.

17 / 44

The 'Dangling `else`'

$$\langle \textit{stmt} \rangle ::= \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle$$
$$\mid\ \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle$$
$$\mid\ \langle \textit{other} \rangle$$

Consider how to parse nested `if` statements e.g.

$$\texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \ \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{other} \rangle\ \texttt{else}\ \langle \textit{other} \rangle$$

To which of the `if`s does the `else` belong to? **Fix:**

$$\langle \textit{stmt} \rangle ::= \langle \textit{matched} \rangle \mid \langle \textit{unmatched} \rangle$$
$$\langle \textit{matched} \rangle ::= \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{matched} \rangle\ \texttt{else}\ \langle \textit{matched} \rangle$$
$$\mid \langle \textit{other} \rangle$$
$$\langle \textit{unmatched} \rangle ::= \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{stmt} \rangle$$
$$\mid \texttt{if}\ \langle \textit{expr} \rangle\ \texttt{then}\ \langle \textit{matched} \rangle\ \texttt{else}\ \langle \textit{unmatched} \rangle$$

Limitations of CFGs

A Context Sensitive Grammar

There are simple languages which are not **context–free**. Here is one:

$$\{a^m \mid m \text{ is a positive power of } 2\}$$

This language is generated by the grammar $G_3 = \{\{a\}, \{S, N, Q, R\}, P, S\}$ with productions:

$$\begin{array}{rlrl} S & ::= QNQ & QN & ::= QR \\ RN & ::= NNR & RQ & ::= NNQ \\ N & ::= a & Q & ::= \epsilon \end{array}$$

When the left–hand side of productions in a grammar contains a sequence of nonterminals, the grammar is **context–sensitive**.

Are CFGs good enough?

We are only studying CFGs. Can CFGs generate the types of features commonly found in programming languages? CFGs are **mostly** adequate. Consider these problems in Pascal:

- type errors:

```
var x: integer; begin x:='c'; ...
```

- procedure arguments

```
procedure p(i,j: integer); ...
p(3,4,5,6);
```

- array indexing

```
var A[1..10] of integer; ... A[2,3]:=0;
```

21 / 44

Are CFGs good enough? p.2

It is **not possible** to write a CFG that generates ***all*** legal Pascal programs **but none** with these types of faults.

A **type–0 grammar** to do this can be devised but it would be non–intuitive and non–transparent, and it would require a Turing machine for a recognizer. Viz. grammar G_3.

Despite these limitations, CFGs are used in practice. A CFG generates a **superset** of a programming language; that's why we actually supplement a CFG with **actions,** which a parser performs to address errors such as those mentioned.

A CFG with actions is known as an **attribute grammar**.

22 / 44

Introducing Parsing

Parsing is the process of determining if a given string of tokens can be generated by a particular grammar. During this process, a parse tree is constructed.

A parser can be constructed for any grammar, but in practice the grammars that are used take a special form.

- **for any CFG** there is a parser that takes at most $O(n^3)$ time to parse a string of n tokens.
- **linear algorithms** suffice, however, to parse essentially all grammars that arise in practice.
- the most common type of programming language parser makes a single left-to-right scan of the input, looking ahead one token at a time. This is known as an **LL**(**1**) parser.

24 / 44

Parsing Methods

Two key classes of methods, differing in the way in which they build the parse tree:

top–down start at the root of the tree and proceed toward the leaves;

bottom–up start at the leaves and work upward to the root.

Thus, a top–down parser seeks the **leftmost derivation** for a sentence, and a bottom–up parser seeks the **rightmost derivation**.

The top–down method allows one to build efficient parsers manually.

The bottom–up method is less intuitive, but it handles a larger class of grammars and enjoys wide tool support.

25 / 44

Top–Down Parsing in General

26 / 44

The Top–Down Technique

The basic algorithm for the top–down parsing technique is as follows:

Start at the root of the parse tree, labelled with the start symbol S and repeat the following:

1. At node n, labelled with non–terminal A, **select one of the productions** for A and **construct children** at n for the symbols on the right–hand side of the production.

2. Find the next node at which a subtree is to be constructed.

The current token being scanned in the input is known as the **lookahead** symbol — initially it is the first token in the input string.

27 / 44

Detailed Example

Here is a grammar for a subset of Pascal types:

```
  ⟨type⟩ ::= ⟨simple⟩
           | "↑" "id"
           | "array" "[" ⟨simple⟩ "]" "of" ⟨type⟩
⟨simple⟩ ::= "integer"
           | "char"
           | "num" ".." "num"
```

Thus, the type of an integer array is:

```
array [integer] of integer
```

We do not distinguish here between `array[5] of integer` and `array[10] of integer`.

28 / 44

Detailed Example

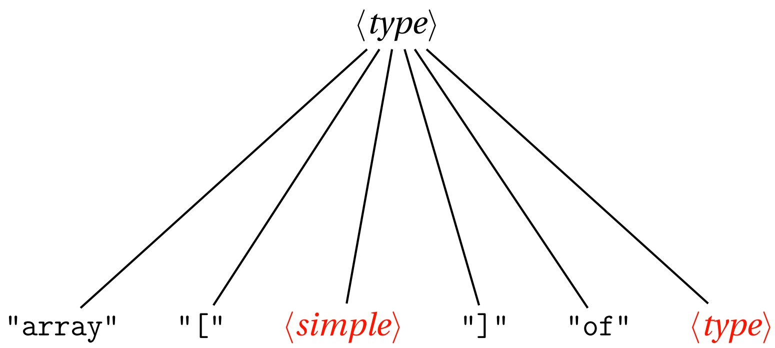
Here is a Pascal type:

```
array [num..num] of integer
```

We are going to parse this expression top–down.

1. Initially the lookahead symbol is the token `"array"`. The root of the parse tree is set to the start symbol ⟨*type*⟩.
2. We find the (only) production which starts with `"array"` and build nodes for its right–hand side, namely for the tokens `"array"`, `"["`, ⟨*simple*⟩, `"]"`, `"of"`, and ⟨*type*⟩.

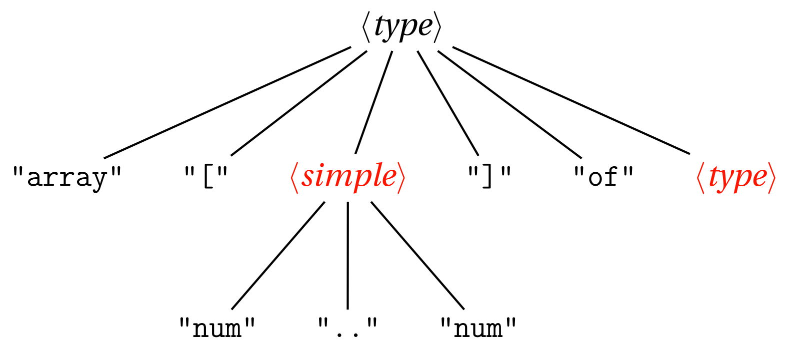
The Parse Tree



Detailed Example

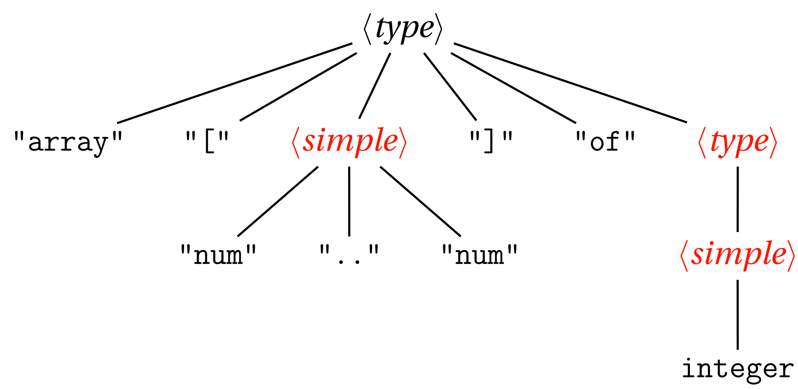
3. We select the leftmost child as the next node to consider. It **matches the lookahead symbol** and it is a **terminal**, so we ***advance*** to the next token in both the tree and the input.
4. The next child is `"["` and the lookahead symbol becomes `"["`. As before, we advance to the next token in both tree and input.
5. We reach the node for ⟨*simple*⟩ and we try each production until we match the lookahead symbol, which is `"num"`. So we apply the production ⟨*simple*⟩ ::= `"num" ".." "num"` and generate three children for this node.

The Parse Tree



The Final Parse Tree

The procedure stops when all children are terminals. This is the final parse tree.



A Predictive Parser

Backtracking is not always necessary; there is a technique in which it does not occur. **Predictive parsing** is a special case of what is known as **recursive–descent parsing,** which involves calling certain procedures recursively to process the input.

We will now program a predictive parser for the grammar of Pascal types.

```
// Part 1/3:
procedure match( t: token )
begin
    if (lookahead = t) then
        lookahead := next\_token();
    else error;
end;
```


A Predictive Parser

```
// Part 2/3:
procedure type();
begin
    if lookahead is in {"integer","char","num"}
       then simple();
    else if lookahead="^" then
        begin match("^"); match("id"); end
    else if lookahead="array" then begin
        match("array"); match("["); simple();
        match("]"); match("of"); type();
        end
    else error();
end;
```

A Predictive Parser

```
// Part 3/3:
procedure simple();
begin
    if lookahead="integer" then
        match("integer");
    else if lookahead="char" then
        match("char");
    else if lookahead="num" then begin
        match("num"); match(".."); match("num");
        end
    else error();
end;
```

37 / 44

Using the Predictive Parser

- To launch the predictive parser, we call the procedure `type()` for the start symbol ⟨*type*⟩. Initially the `lookahead` variable is set to `"array"`, which is the first token in the input.
- The `match(t)` procedure moves by one token in the input whenever the lookahead symbol matches the current node in the parse tree.
- Predictive parsing relies on information about what first symbols appear on the right–hand side of a production. For this grammar, there is exactly one possibility at each point during the parse.

38 / 44

The `FIRST` Set

Formally, a predictive parser uses what is known as the `FIRST` set or **starter set**.

- For a right–hand side α of a production $A ::= \alpha$, we define $\texttt{FIRST}(\alpha)$ as the set of tokens that appear as the first symbols of one or more strings generated from α.
- If $\alpha = \epsilon$ or α can generate ϵ, then ϵ also belongs to $\texttt{FIRST}(\alpha)$.

$$
\begin{aligned}
&\texttt{FIRST}(\langle \textit{simple} \rangle) = \{\texttt{"integer"}, \texttt{"char"}, \texttt{"num"}\} \\
&\texttt{FIRST}(\texttt{"}\uparrow\texttt{" "id"}) = \{\texttt{"}\uparrow\texttt{"}\} \\
&\texttt{FIRST}(\texttt{"array" "[" } \langle \textit{simple} \rangle \texttt{ "]" "of" } \langle \textit{type} \rangle) = \\
&\qquad = \{\texttt{"array"}\}
\end{aligned}
$$

39 / 44

The `FIRST` Set

If there are two productions such that $A ::= \alpha$ and $A ::= \beta$, then we have to consult the `FIRST` sets of α and β to decide what to do.

- if the lookahead symbol is in $\texttt{FIRST}(\alpha)$, then α is used.
- if the lookahead symbol is in $\texttt{FIRST}(\beta)$, then β is used.

Recursive–descent parsing requires

$$\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) = \emptyset$$

40 / 44

How to Write a Predictive Parser in General

A predictive parser consists of a procedure for **every non–terminal** in a grammar. Each procedure does the following two things:

1. It decides which production to use by looking at the lookahead symbol.

   If the lookahead symbol is in $\texttt{FIRST}(\alpha)$ for some production $A ::= \alpha$ then that production is used; if there is a conflict between two right–hand sides, the method **fails**. If the lookahead symbol does not belong to any `FIRST` set, then a production with ϵ is used.

2. The procedure "uses" a production, or applies it, by "mimicking" the right–hand side in terms of code i.e. by calling the `match(t)` function as many times as necessary.

41 / 44

Some Remarks

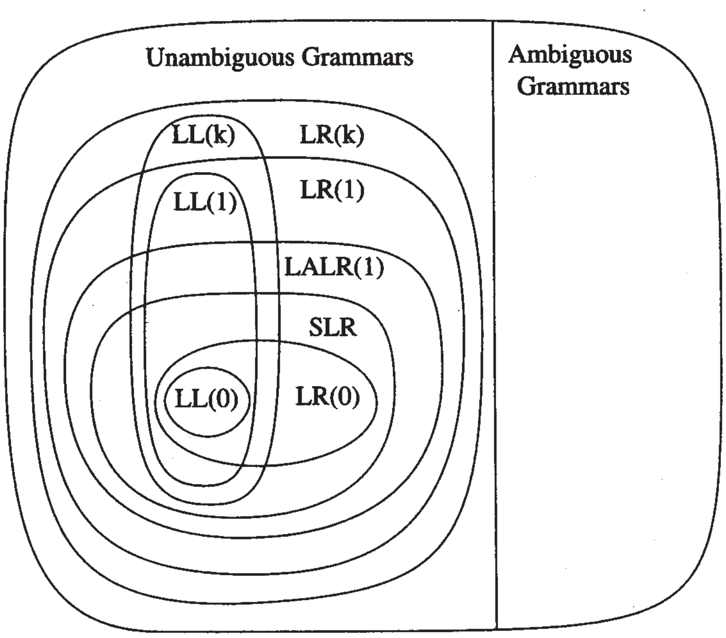
A recursive–descent parser will **loop forever** on a left–recursive grammar! Luckily, left recursion can be eliminated using a well-documented algorithm.

Our predictive parser used **one** lookahead symbol to decide which production to apply. It is an **LL**(**1**) parser, since it reads the input from **L**eft to right and seeks the **L**eftmost derivation.

The parsing algorithm we used only applies to a limited group of grammars, known unsurprisingly as **LL**(**1**) grammars. This is the most common type of grammar, but there are many other classes of grammars for which parsing algorithms are known.

42 / 44

Hierarchy of Grammar Classes



43 / 44

Summary

- We have studied grammars in detail, looking at parse trees, recursion and ambiguity.
- We considered whether CFGs are adequate for specifying programming language syntax.
- We looked at the top–down parsing process in detail and implemented a predictive **LL**(**1**) parser.

Next and final lecture: **Bottom–up parsing**