# Introduction to Lexing and Parsing Techniques[a]

Nick Papanikolaou

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

Lecture 1: **Lexical Analysis**

---

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

# Introduction

## Putting Things in Context

- Study of automata, grammars and languages is relevant to the design and implementation of **compilers** and **interpreters.**

- These lectures will be concerned with the processes of **lexing** and **parsing,** which are the main phases of any compiler.

- We will discuss both the relevant **theory**, and how to use lexer and parser generators in **practice**.

## Outline

1. The Chomsky Hierarchy; Recognizers
2. Translators and Compilers
3. Lexical Analysis
4. Using Scanner Generators

# The Chomsky Hierarchy

## Chomsky Hierarchy

- Noam Chomsky introduced a classification for grammars and the languages they generate, proposing they may provide an adequate model for natural languages.

- The classification imposes restrictions on the forms of production a grammar may have.

- Four types of grammar:
  - **Unrestricted Grammars** (Type 0)
  - **Context Sensitive Grammars** (Type 1)
  - **Context–Free Grammars** (Type 2)
  - **Regular Grammars** (Type 3)

## Chomsky Hierarchy: Restrictions

**Unrestricted Grammars** may have productions of the form $\alpha \Rightarrow \beta$ where $\alpha$ and $\beta$ are arbitrary strings of symbols with $\alpha \neq \epsilon$.

**Context Sensitive Grammars** have productions of the form $\alpha \Rightarrow \beta$ such that $\beta$ is at least as long as $\alpha$.

**Context–Free Grammars** have productions of the form $A \Rightarrow \alpha$, where $\alpha$ is a sequence of variable or terminal symbols.

**Regular Grammars** are either:

**left–linear grammars** which have productions of the form $A \Rightarrow wB$ or $A \Rightarrow w$ only; or

**right–linear grammars** which have productions of the form $A \Rightarrow Bw$ or $A \Rightarrow w$ only,

where $w$ is a possibly empty string of terminals.

## Chomsky Hierarchy: Theorem

The classes of grammars correspond to four types of languages (Note: R.E.Ls. = Recursively Enumerable Languages):

$$\begin{array}{rcl}
\text{unrestricted grammars (Type 0)} & \leftrightarrow & \text{R.E. Ls.} \\
\text{context sensitive grammars (Type 1)} & \leftrightarrow & \text{C.S. Ls.} \\
\text{context–free grammars (Type 2)} & \leftrightarrow & \text{C.F. Ls.} \\
\text{regular grammars (Type 3)} & \leftrightarrow & \text{R.Ls.}
\end{array}$$

**Hierarchy Theorem.** *The Type-i languages properly include the Type-(i+1) languages for i=0, 1, and 2.*

In other words,

$$\boxed{\text{Type 0 Ls.} \supset \text{Type 1 Ls.} \supset \text{Type 2 Ls.} \supset \text{Type 3 Ls.}}$$

### Recognizers

A **recognizer** is a machine which accepts a given language. The recognizers corresponding to the different classes of languages in the Chomsky Hierarchy are:

R.E.Ls. (Type 0) ↔ Turing Machines
C.S.Ls. (Type 1) ↔ Linear Bounded Automata
C.F.Ls. (Type 2) ↔ Pushdown Automata
R.Ls. (Type 3) ↔ Finite Automata

The ↔ indicates an **equivalence** between the classes of languages shown and the corresponding machines that recognize them. *This is studied in more detail elsewhere in the course.*

## Translators and Compilers

### Overview

- The implementation of a translator involves programming, or generating code for, a deterministic finite automaton known as a **lexer**, and a pushdown automaton known as a **parser**.

- Programming languages generally lie in the class of **C.F.Ls.**; we will restrict our attention to **regular grammars** which are relevant to lexing, and **context–free grammars,** used for parsing.

### Translators

A **translator** is a tool that inputs a program in a **source language** and converts it into an object program in a **target language.**

A **compiler** is a translator from a high–level source language to a low–level target language.

The tasks of any compiler form part of two processes: **analysis** and **synthesis**. We will be concerned only with analysis, which is divided into stages:

1. **lexical analysis,** (Lecture 1)
2. **syntactic analysis,** (Lectures 2 and 3)
3. **semantic analysis.**

## Lexical Analysis

- A **lexer** separates the source program into pieces known as **tokens**.
- It reads the source program one character at a time.
- It handles issues such as whitespace and statements spread over many lines.
- Often a lexer stores constants, labels and variable names in a **symbol table**.
- A lexer outputs, for each token $t_i$, a pair $(\gamma_i, N(t_i))$, where $N(t_i)$ is an integer representing the token internally, and $\gamma_i$ is the address of the token in the symbol table.

## Syntactic Analysis

- A **parser** groups the tokens produced by the lexer into larger syntactic classes, e.g. expressions, statements, procedures.
- It outputs a **syntax tree,** whose leaves are tokens and all nonleaves are syntactic class types.
- A parser uses the grammar of the language in which the program is expressed to determine what the syntactic classes are.

## Semantic Analysis

Semantic analysis involves interpreting the meaning of the source program; the semantic analyzer typically reduces expressions to some **intermediate representation**.

**Example 1.** *The expression*

`(A+B)*(C+D)`

*is transformed by the semantic analyzer into the following triple:*

$$\Big((+, A, B, T_1), (+, C, D, T_2), (*, T_1, T_2, T_3)\Big)$$

# Lexical Analysis in Depth

## The Lexer (aka Scanner)

- A scanner produces a stream of **tokens** or **lexical units** from the source program.
- Tokens are actually the **terminal symbols** of the grammar of the source language.
- The scanner's secondary functions are (typically):
  - to eliminate **whitespace** (tabs, blanks, comments);
  - to find **lexical errors** (e.g. misspellings of keywords);
  - to store certain classes of tokens in a **symbol table**;
  - to determine the **types** of tokens.

## Example Output of a Lexer

```
SUM: A = A + B;
     GOTO DONE;
```

Output of lexer for each token $t_i$ is $(\gamma_i, N(t_i))$ as shown:

```
SUM     1     3
:       0     11
A       2     1
=       0     6
A       2     1
+       0     5
B       3     1
;       0     12
GOTO    0     4
DONE    4     3
;       0     12
```

The string constant corresponding to token $t_i$ is known as a **lexeme**. Each lexeme is represented by an internal **token number** $N(t_i)$.

## Why Lexing and Parsing are Separate

There are two ways of integrating lexer and parser:

1. implementing lexer **as a separate pass,** producing a big output file or token list in memory;
2. implementing lexer **as a function** `next_token()`**,** called from the parser whenever it is needed.

The second way is used more often in practice.

However, the advantages of treating lexing as a separate pass are:

- reading one character at a time from disk can be slow, so doing it in one go makes parsing faster;
- the lexer makes more information available to the parser (viz. symbol tables).

## Tokens

There is no completely general rule for deciding what constitutes a token and what does not, given a particular grammar. The following are the most common kinds of tokens:

- **keywords,** e.g. `if`, `then`, `else`, `goto`, . . .
- **identifiers,** e.g. `myconst`, . . .
- **constants,** e.g. `1`, `2`, `true`, . . .
- **operators,** e.g. `+`, `*`, `&&`, . . .
- **delimiters,** e.g. `(`, `)`, {, }, . . .

## Describing Tokens

One way of specifying the tokens in a programming language is to use a **regular grammar.**

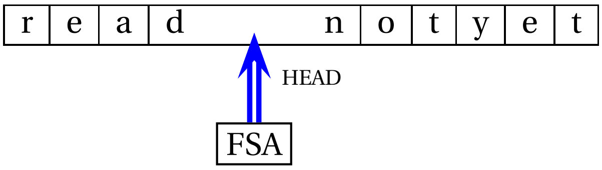
**Example 2** (Natural Numbers)**.**

$$\begin{aligned} \langle \textit{unsigned int} \rangle ::= {} & 0 \\ & \mid \vdots \\ & \mid 9 \\ & \mid 0\langle \textit{unsigned int}\rangle \\ & \mid \vdots \\ & \mid 9\langle \textit{unsigned int}\rangle \end{aligned}$$

## Describing Tokens cont.

- A regular grammar is a **generative** means of specifying tokens.
- For our purposes, a **recognitive** means of describing tokens is desirable.
- Describing tokens in terms of how they can be recognized, or accepted, is done using **finite–state automata** (FSA).

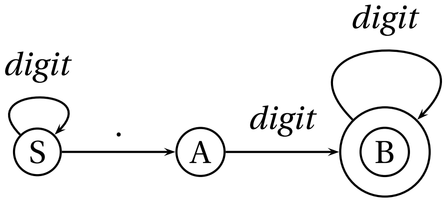
FSA reads an input tape one character at a time and changes internal state depending on character read:



## Describing Tokens using FSA

FSA are expressed using **transition diagrams**.

Here is a transition diagram for an FSA which accepts **decimal real numbers** with at least one digit after the decimal point.



## Implementing a Lexer/Scanner

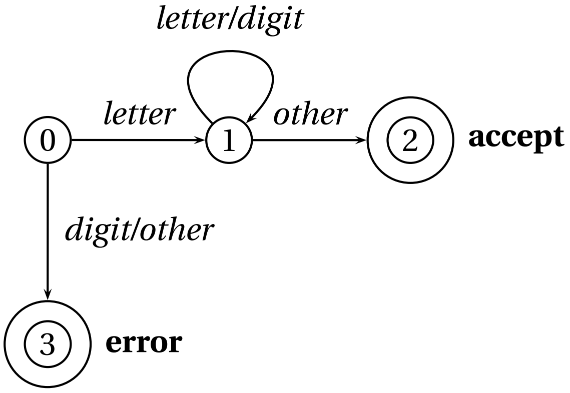
Suppose we want to implement a lexer for **identifiers** in a programming language.

Identifiers consist of *at least one letter*, followed optionally by *one or more digits*. Here is the grammar for identifiers:

$$\begin{aligned} \langle \mathit{letter} \rangle &::= \mathtt{a} \mid \cdots \mid \mathtt{z} \mid \mathtt{A} \mid \cdots \mid \mathtt{Z} \\ \langle \mathit{digit} \rangle &::= \mathtt{0} \mid \cdots \mid \mathtt{9} \\ \langle \mathit{ident} \rangle &::= \langle \mathit{letter} \rangle \; ( \langle \mathit{letter} \rangle \mid \langle \mathit{digit} \rangle ) \end{aligned}$$

## DFA for Identifiers

The DFA corresponding to the grammar for identifiers is as follows:



## Implementing the DFA

To implement the lexer for identifiers, two structures are used: the **character class map** and the **transition table:**

| Character | `a--z` | `A--Z` | `0--9` | *other* |
|---|---|---|---|---|
| Class | *letter* | *letter* | *digit* | *other* |

The transition table is a representation of the FSA state changes. The table contains the output of the function `next_state(curr_state, toktype)` :

|  | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| *letter* | 1 | 1 | – | – |
| *digit* | 3 | 1 | – | – |
| *other* | 3 | 2 | – | – |

## Code for the lexer

```
char := next_char();
curr_state := 0;
done := false;
token_value := "";
while (not done) {
    class := char_class[ char ];
    curr_state := next_state[ curr_state, class ];
    switch (state) {
        case 1: /* still reading an identifier */
            token_value := token_value + char;
            char := next_char;
            break;
```

### Code for the lexer p. 2

```
        case 2: /* accept state */
            token_type := IDENTIFIER;
            done := true;
            break;
        case 3: /* error */
            token_type := ERROR;
            done := true;
            break;
    }
}
return token_type;
```

**Key point:** DFA implemented simply as a `case` statement on the current state.

## Using Scanner Generators

### Scanner Generators

Scanner generators automatically construct code from regular expression–like descriptions. They:

- construct a DFA;
- apply state minimization techniques to reduce the DFA to the smallest possible one;
- output code for the scanner.

A key issue in code generation is handling the interface with the parser.

## Regular Expressions

| | |
|---|---|
| a | An ordinary character stands for itself. |
| $\epsilon$ | The empty string. |
| $M \mid N$ | Alternation; choosing M or N. |
| $M \cdot N$ | Concatenation; M followed by N. |
| $M^*$ | Repetition of M **zero** or more times. |
| $M^+$ | Repetition of M **one** or more times. |
| $M?$ | Option; **zero or one** occurrence of M. |
| `[a-zA-Z]` | Character set alternation. |
| . | Any single character except newline. |
| `"**man!"` | Quotation; a string literal. |

Remember: regular expressions and regular grammars are **equivalent;** they both generate the class of **regular languages.**

## Regular Expressions for Tokens

Here are some examples of regular expressions describing tokens in a programming language.

| | |
|---|---|
| ⟨*if*⟩ | `if` |
| ⟨*ident*⟩ | `[a-z][a-z0-9]*` |
| ⟨*num*⟩ | `[0-9]+` |
| ⟨*real*⟩ | `([0-9]+"."[0-9]*)\|([0-9]*"."[0-9]+)` |
| Comments | `("//"[a-zA-Z]*"\n")` |

## `lex`: A Scanner Generator

- `lex` is a well-known and widely supported scanner generator designed by AT&T in the 70s; it was traditionally bundled with UNIX.
- `flex` is an improved, faster version of `lex` and is compatible with `lex` input files.
- **Input** to `lex` is a file containing a definition of the tokens in a language; tokens are defined using **regular expressions.**
- For each token there must be a corresponding **action,** specifying how the lexer should handle it.
- The **output** of `lex` is a **C** program containing an implementation of the lexer; the lexer is called through the `yylex()` function.
- Note: `lex` is designed to be used with the `yacc` parser generator.

## `lex`: File Structure

A `lex` input file has the following structure in general:

Part A (comments, C code, mnemonics, start states)
%%
Part B (rules defining tokens and actions)
%%
Part C (C code including `main(...)`)

Part A usually contains definitions of token numbers, e.g.

```
#define T_id 1
#define T_realconst 2
```

etc.

## `lex`: An example

```
%{                              //PART A
#define T_eof   0
#define T_id    1
#define T_real  2  ...
#define T_while 52
%}
%%                              //PART B
"and"       { return T_and; } ...
":="        { return T_assign; }
"while"     { return T_while; }
// Regexp for real numbers:
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)
            { return T_real; }
%%                              //PART C ...
```

## `lex`: An example p.2

```
%%                          //PART C continued
void main() {
    int token;
    do {
        token = yylex();
        printf("token=%d, lexeme=\"%s\"\n",
                token, yytext);
    } while (token != T_eof);
}
```

This code will generate a lexer which **prints out each token** found in the input. Normally, additional code for handling **errors** should be added.

--→ Play with `lex/flex` at home. ←--

## Introducing `javacc` and `sablecc`

- `javacc` and `sablecc` are tools which generate lexical analyzers and parsers in **Java**.
- Each of these tools has a different format for **lexical specifications**, as we shall see.
- To generate parsers with these tools, one must supply a complete grammar. But whether you're using `javacc` or `sablecc`, both lexical specs and grammar go in the same file.
- We will look at the syntax for lexical specifications corresponding to the token types considered on slide 12, namely $\langle \mathit{if} \rangle$, $\langle \mathit{ident} \rangle$, $\langle \mathit{num} \rangle$ and $\langle \mathit{real} \rangle$.

## `javacc` Syntax for lexer

```
/* Java compilation unit with class decl. */
PARSER_BEGIN(MyParser)
    class MyParser {}
PARSER_END(MyParser)

/* For the regexp on the right,
 * return token on the left. */
TOKEN : {
    < IF: "if" >
  | < #DIGIT: ["0"-"9"] >
  | < ID: ["a"-"z"] (["a"-"z"] | <DIGIT>)* >
  | < NUM: (<DIGIT>)+ >
  | < REAL: ( (<DIGIT>+ "." (<DIGIT>)* ) |
            ( (<DIGIT>)* "." (<DIGIT>)+ ) >
}
```

## `javacc` Syntax for lexer p.2

```
/* definitions of things to skip */
SKIP : {
      <"//" (["a"-"z"])* ("\n" | "\r" | "\r\n")>
    | " " | "\t" | "\n"
}

/* Standard code for main method */
void Start() :
{}
{ ( <IF> | <ID> | <NUM> | <REAL> )* }
```

## `sablecc` Syntax for the same lexer

```
Helpers
    digit = [’0’..’9’];
Tokens
    if = ’if’;
    id = [’a’..’z’]([’a’..’z’] | (digit))*;
    number = digit+;
    real = ((digit)+ ’.’ (digit)*) |
           ((digit)* ’.’ (digit)+);
    whitespace = (’ ’ | ’\t’ | ’\n’)+;
    comments = (’//’ [’a’..’z’]* ’\n’);
Ignored Tokens
    whitespace, comments;
```

# Review

## Summary

In this lecture we have discussed:

- the **hierarchy** of grammars and languages they generate;
- the fact that **regular grammars** and **CFGs** are most useful in practice, and are the basis for **lexical** and **syntactic** analysis of programming languages respectively.
- the **correspondence** between different types of grammars and the automata that recognize, or accept them.
- the **analytical** processes in a compiler.

## Summary p. 2

We also discussed:

- the overall functions of a **lexer/scanner;**
- the representation of **tokens** using **finite automata**, **regular grammars** and **regular expressions;**
- how a DFA corresponding to a lexer may be implemented using a set of `case` statements;
- how to use `lex`, `javacc` and `sablecc`.

## Next Lecture

The next lecture will introduce the parsing problem, which deals with recognizing context–free languages.

- We will have some more to say on **grammars** and their properties; as we shall see, parsing is more difficult than lexing.
- We will find out whether context–free grammars are really adequate to model **programming language syntax;**
- We will see what a **context–sensitive grammar** actually looks like;