

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)

**Next:** [2.1 Regular Expressions \(REs\)](#)
**Up:** [CSE 5317/4305: Design and](#)
**Previous:** [1.3 Compiler Architecture](#)  
[Contents](#)

## 2 Lexical Analysis

A *scanner* groups input characters into tokens. For example, if the input is

```
x = x*(b+1);
```

then the scanner generates the following sequence of tokens

```
id(x)
=
id(x)
*
(
id(b)
+
num(1)
)
;
```

where `id(x)` indicates the identifier with name `x` (a program variable in this case) and `num(1)` indicates the integer 1. Each time the parser needs a token, it sends a request to the scanner. Then, the scanner reads as many characters from the input stream as it is necessary to construct a single token. The scanner may report an error during scanning (eg, when it finds an end-of-file in the middle of a string). Otherwise, when a single token is formed, the scanner is suspended and returns the token to the parser. The parser will repeatedly call the scanner to read all the tokens from the input stream or until an error is detected (such as a syntax error).

Tokens are typically represented by numbers. For example, the token `*` may be assigned number 35. Some tokens require some extra information. For example, an identifier is a token (so it is represented by some number) but it is also associated with a string that holds the identifier name. For example, the token `id(x)` is associated with the string, `"x"`. Similarly, the token `num(1)` is associated with the number, 1.

Tokens are specified by patterns, called *regular expressions*. For example, the regular expression `[a-z][a-zA-Z0-9]*` recognizes all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.

A typical scanner:

1. recognizes the *keywords* of the language (these are the reserved words that have a special meaning in the language, such as the word `class` in Java);
2. recognizes special characters, such as `(` and `)`, or groups of special characters, such as `:=` and `==`;
3. recognizes identifiers, integers, reals, decimals, strings, etc;
4. ignores whitespaces (tabs and blanks) and comments;
5. recognizes and processes special directives (such as the `#include "file"` directive in C) and macros.

A key issue is speed. One can always write a naive scanner that groups the input characters into lexical words (a lexical word can be either a sequence of alphanumeric characters without whitespaces or special characters, or just one special character), and then tries to associate a token (ie. number, keyword, identifier, etc) to this lexical word by performing a number of string comparisons. This becomes very expensive when there are many keywords and/or many special lexical patterns in the language. In this section you will learn how to build efficient scanners using regular expressions and finite automata. There are automated tools called *scanner generators*, such as *flex* for C and *JLex* for Java, which construct a fast scanner automatically according to specifications (regular expressions). You will first learn how to specify a scanner using regular expressions, then the underlying theory that scanner generators use to compile regular expressions into efficient programs (which are basically finite state machines), and then you will learn how to use a scanner generator for Java, called JLex.

---

## Subsections

- [2.1 Regular Expressions \(REs\)](#)
- [2.2 Deterministic Finite Automata \(DFAs\)](#)
- [2.3 Converting a Regular Expression into a Deterministic Finite Automaton](#)
- [2.4 Case Study: The Calculator Scanner](#)

---

<a href="#">Next</a>	<a href="#">Up</a>	<a href="#">Previous</a>	<a href="#">Contents</a>
----------------------	--------------------	--------------------------	--------------------------

**Next:** [2.1 Regular Expressions \(REs\)](#) **Up:** [CSE 5317/4305: Design and](#) **Previous:** [1.3 Compiler Architecture](#)  
[Contents](#)

*fegaras 2012-01-10*