Lexical and Syntax Analysis
(of Programming Languages)

Introduction
Lexical and Syntax Analysis
(of Programming Languages)

Introduction
What is Parsing?

A parser also checks that the input string is **well-formed**, and if not, rejects it.
What is Parsing?

A parser also checks that the input string is **well-formed**, and if not, rejects it.
Example 1

CSV (Comma Separated Value)

Parser

Array of pairs

Charlton, 49
Lineker, 48
Beckham, 17
Example 1

Charlton, 49
Lineker, 48
Beckham, 17

Parser

CSV (Comma Separated Value)

Array of pairs

“Charlton”
49

“Lineker”
48

“Beckham”
17
Data structure?

A **data structure** is typically a **data type** in some programming language, e.g. C.

The type of a player:

```c
typedef struct {
    char* name;
    int goals;
} Player;
```

The type of a squad of players:

```c
typedef struct {
    Player* players;
    int size;
} Squad;
```
A **data structure** is typically a value of a **data type** in some programming language, e.g. C.

The type of a player:

```c
typedef struct {
    char* name;
    int goals;
} Player;
```

The type of a squad of players:

```c
typedef struct {
    Player* players;
    int size;
} Squad;
```
Why data structures?

Data structures are convenient to process by a computer program.

The total goals scored by all players in a squad:

```c
int total(Squad s) {
    int i, sum = 0;
    for (i = 0; i < s.size; i++)
        sum += s.players[i].goals;
    return sum;
}
```
Why data structures?

Data structures are convenient to process by a computer program.

The total goals scored by all players in a squad:

```c
int total(Squad s)
{
    int i, sum = 0;
    for (i = 0; i < s.size; i++)
        sum += s.players[i].goals;
    return sum;
}
```
Example 1: the problem

We want to be able to **parse** CSV files to values of type *Squad* so we can process them **conveniently**.

```c
Squad parse(char* input)
{
    ...
}
```

LSA will teach you how to fill in the dots. (This is a rather easy example, though!)
Example 1: the problem

We want to be able to parse CSV files to values of type *Squad* so we can process them conveniently.

```
Squad parse(char* input)
{
    ...
}
```

LSA will teach you how to fill in the dots. (This is a rather easy example, though!)
Everyday parsing

- Our **email clients** parse email headers, allowing search by *to* & *from* address etc.

- Our **web browsers** parse HTML, JavaScript, CSS, etc.

- Our copies of **Call of Duty** parse configuration files and saved-game states.
Everyday parsing

- Our email clients parse email headers, allowing search by to & from address etc.
- Our web browsers parse HTML, JavaScript, CSS, etc.
- Our copies of Call of Duty parse configuration files and saved-game states.
LSA of PLs

- In LSA, we are interested in parsing in general.
- But we have a special interest in parsing programming languages (PLs). Why?
  - “If we can parse a PL, we can parse anything.” 😊
- In practice, we often want to parse PL-like languages.
- Preparation for CGO.
LSA of PLs

- In LSA, we are interested in parsing in general.
- But we have a special interest in parsing programming languages (PLs). Why?
  - “If we can parse a PL, we can parse anything.” 😊
- In practice, we often want to parse PL-like languages.
- Preparation for CGO.
Example 2

A pascal statement

Parser

foo := 20 + bar

An abstract syntax tree

We will return to this example later!
Example 2

foo := 20 + bar

A pascal statement

Parser

An abstract syntax tree

We will return to this example later!
LSA & CGO

The connection between *Lexical and Syntax Analysis* (2\textsuperscript{nd} year module) and *Code Generation and Optimisation* (3\textsuperscript{rd} year module).
LSA & CGO

The connection between *Lexical and Syntax Analysis* (2\textsuperscript{nd} year module) and *Code Generation and Optimisation* (3\textsuperscript{rd} year module).
LSA & CGO

Easy for humans to write and understand

Source Program (String)

Easy for compiler to process

Source Program (Data structure)

Easy for machines to execute

Target Program

LSA

CGO
LSA & CGO

Easy for **humans** to **write and understand**

Easy for **compiler** to **process**

Easy for **machines** to **execute**

Source Program (String)

Source Program (Data structure)

Target Program

LSA

CGO
OTHER MOTIVATIONS

For studying parsing
OTHER MOTIVATIONS

For studying parsing
To understand how computers work

- Computer Architecture
- Programming Languages
- LSA & CGO
To understand how computers work

- Computer Architecture
- LSA & CGO
- Programming Languages
To practice applying your knowledge

[Diagram showing POP, TOC, TAD merging into LSA]
To practice applying your knowledge
PARSING

= 

LEXICAL ANALYSIS  
+ 
SYNTAX ANALYSIS
PARSING

= 

LEXICAL ANALYSIS

+ 

SYNTAX ANALYSIS
Lexical Analysis

(Also known as “scanning”)

- Identifies the **lexemes** in a sentence.

- **Lexeme**: a minimal meaningful unit of a language.

- Converts each lexeme to a **token**.

- Throws away ignorable text such as spaces, new-lines, and comments.
Lexical Analysis
(Also known as “scanning”)

- Identifies the **lexemes** in a sentence.

- **Lexeme**: a minimal meaningful unit of a language.

- Converts each lexeme to a **token**.

- Throws away ignorable text such as spaces, new-lines, and comments.
What is a token?

- Every token has an **identifier**, used to denote the **kind** of lexeme that it represents, e.g.

<table>
<thead>
<tr>
<th>Token identifier</th>
<th>denotes a</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>+ operator</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>:= operator</td>
</tr>
<tr>
<td>VAR</td>
<td>variable</td>
</tr>
<tr>
<td>NUM</td>
<td>number</td>
</tr>
</tbody>
</table>

- **Some** tokens have a **component value**, conventionally written in parenthesis after the identifier, e.g. `VAR(foo), NUM(12)`. 
What is a token?

- Every token has an identifier, used to denote the kind of lexeme that it represents, e.g.

<table>
<thead>
<tr>
<th>Token identifier</th>
<th>denotes a</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>+ operator</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>:= operator</td>
</tr>
<tr>
<td>VAR</td>
<td>variable</td>
</tr>
<tr>
<td>NUM</td>
<td>number</td>
</tr>
</tbody>
</table>

- Some tokens have a component value, conventionally written in parenthesis after the identifier, e.g. \(VAR(\text{foo}),\ NUM(12)\).
Lexical Analysis

Example input:

\[ foo := 20 + bar \]

Example output:

\[ VAR(foo), ASSIGN, NUM(20), PLUS, VAR(bar) \]
Lexical Analysis

Stream of characters → Stream of tokens

Example input:

\[\text{foo} := 20 + \text{bar}\]

Example output:

\[\text{VAR(foo)}, \text{ASSIGN}, \text{NUM(20)}, \text{PLUS}, \text{VAR(bar)}\]
Lexical Analysis

Lexemes are specified by regular expressions. For example:

\[
\begin{align*}
\text{digit} & = 0 \mid \ldots \mid 9 \\
\text{letter} & = a \mid \ldots \mid z \\
\text{number} & = \text{digit} \cdot \text{digit}^* \\
\text{variable} & = \text{letter} \cdot (\text{letter} \mid \text{digit})^*
\end{align*}
\]

Example numbers:

1 4 43 634

Example variables:

x foo foo2 x1y20
Lexical Analysis

Lexemes are specified by regular expressions. For example:

\[
\begin{align*}
\text{digit} & = 0 \mid \ldots \mid 9 \\
\text{letter} & = a \mid \ldots \mid z \\
\text{number} & = \text{digit} \cdot \text{digit}^* \\
\text{variable} & = \text{letter} \cdot (\text{letter} \mid \text{digit})^*
\end{align*}
\]

Example numbers:
1
4
43
634

Example variables:
x
foo
foo2
x1y20
Syntax Analysis
(Also known as “parsing”)

- **Syntax**: the set of rules defining valid strings of a language.

- Syntax analysis converts a stream of symbols to a parse tree:
  - a **proof** that a given input is valid according to the language syntax;
  - also a **structure-rich** representation of the input that is convenient to process.
Syntax Analysis
(Also known as “parsing”)

- **Syntax:** the set of rules defining valid strings of a language.

- Syntax analysis converts a stream of symbols to a **parse tree:**
  - a **proof** that a given input is valid according to the language syntax;
  - also a **structure-rich** representation of the input that is convenient to process.
Syntax Analysis

- The syntax of a language is usually specified by a grammar.

- Example:

  \[
  \begin{align*}
  \text{expr} & \rightarrow \text{VAR}(v) \\
  & \mid \text{NUM}(n) \\
  & \mid \text{expr} \text{PLUS} \text{expr} \\
  \text{stmt} & \rightarrow \text{VAR}(v) \text{ASSIGN} \text{expr}
  \end{align*}
  \]

  Where \( v \) represents any variable name and \( n \) any number.
Syntax Analysis

- The syntax of a language is usually specified by a grammar.
- Example:

  | expr      | → | VAR(v) |
  |          | / | NUM(n) |
  |          | / | expr PLUS expr |

  | stmt      | → | VAR(v) ASSIGN expr |

Where \(v\) represents any variable name and \(n\) any number.
Syntax Analysis

Example input:

```
VAR(foo), ASSIGN, NUM(20), PLUS, VAR(bar)
```

Example output:

```
stmt
  VAR(foo) ASSIGN expr
    expr PLUS expr
      NUM(20) VAR(bar)
```
Syntax Analysis

Example input:

VAR(foo), ASSIGN, NUM(20), PLUS, VAR(bar)

Example output:

```
stmt
  |
  V
VAR(foo) ASSIGN expr
    |
    V
expr PLUS expr
      |
      V
NUM(20) VAR(bar)
```
Syntax Analysis subsumes Lexical Analysis

- Any language that can be accepted by a regular expression can be accepted by a grammar.
- But not vice-versa!*
- Hence Syntax Analysis is more powerful than Lexical Analysis.

* Can anyone give a simple example?
Syntax Analysis *subsumes* Lexical Analysis

- Any language that can be accepted by a regular expression can be accepted by a grammar.
- But not vice-versa!*
- Hence Syntax Analysis is more **powerful** than Lexical Analysis.

* Can anyone give a simple example?
Why bother with Lexical Analysis?

- **Convenience**: regular expressions more convenient than grammars to define regular strings.
- **Efficiency**: there are efficient algorithms for matching regular expressions that do not apply in the more general setting of grammars.
- **Modularity**: split a problem into two smaller problems.
Why bother with Lexical Analysis?

- **Convenience**: regular expressions are more convenient than grammars to define regular strings.

- **Efficiency**: there are efficient algorithms for matching regular expressions that do not apply in the more general setting of grammars.

- **Modularity**: split a problem into two smaller problems.
THE LSA MODULE

Structure and content
THE LSA MODULE

Structure and content
Objectives

To learn how to implement efficient parsers

- using a general purpose programming language (C);
- using an automatic parser generator (*Flex & Bison*).

and to learn the **theory** behind parser generation from **regular expressions** and **grammars**.
Objectives

To learn how to implement efficient **parsers**

- using a general purpose programming language (**C**);
- using an **automatic parser generator** (**Flex & Bison**).

and to learn the **theory** behind parser generation from **regular expressions** and **grammars**.
Practicals

- 4 **practicals** in the lab.
- Idea is to **practice** the techniques developed in the lectures.
- Room CSE/069.
- Spring weeks 9 & 10.
- Summer weeks 4 & 5.
- Two practical groups: A and B. Find your group on the LSA web page.

<table>
<thead>
<tr>
<th>Practical</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lexical Analysis</td>
</tr>
<tr>
<td>2</td>
<td><em>Flex</em>, a Lexical Analyser Generator</td>
</tr>
<tr>
<td>3</td>
<td>Recursive Descent Parsing</td>
</tr>
<tr>
<td>4</td>
<td><em>Bison</em>, a Parser Generator</td>
</tr>
</tbody>
</table>

Organisation of practicals
Practicals

- 4 **practicals** in the lab.
- Idea is to **practice** the techniques developed in the lectures.
- Room CSE/069.
- Spring weeks 9 & 10.
- Summer weeks 4 & 5.
- Two practical groups: A and B. Find your group on the LSA web page.

<table>
<thead>
<tr>
<th>Practical</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lexical Analysis</td>
</tr>
<tr>
<td>2</td>
<td><em>Flex</em>, a Lexical Analyser Generator</td>
</tr>
<tr>
<td>3</td>
<td>Recursive Descent Parsing</td>
</tr>
<tr>
<td>4</td>
<td><em>Bison</em>, a Parser Generator</td>
</tr>
</tbody>
</table>

Organisation of practicals
Lecture contents

- 14 lectures, with notes arranged into 9 chapters.
- A single chapter may be covered in less than or more than one lecture.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Abstract Syntax</td>
</tr>
<tr>
<td>3</td>
<td>Lexical Analysis</td>
</tr>
<tr>
<td>4</td>
<td><em>Flex</em>, a Lexical Analyser Generator</td>
</tr>
<tr>
<td>5</td>
<td>Grammars</td>
</tr>
<tr>
<td>6</td>
<td>Recursive Descent Parsing</td>
</tr>
<tr>
<td>7</td>
<td><em>Bison</em>, a Parser Generator</td>
</tr>
<tr>
<td>8</td>
<td>Top-Down Parsing</td>
</tr>
<tr>
<td>9</td>
<td>Bottom-Up Parsing</td>
</tr>
</tbody>
</table>

Organisation of Lectures
Lecture contents

- 14 lectures, with notes arranged into 9 chapters.
- A single chapter may be covered in less than or more than one lecture.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Abstract Syntax</td>
</tr>
<tr>
<td>3</td>
<td>Lexical Analysis</td>
</tr>
<tr>
<td>4</td>
<td><em>Flex</em>, a Lexical Analyser Generator</td>
</tr>
<tr>
<td>5</td>
<td>Grammars</td>
</tr>
<tr>
<td>6</td>
<td>Recursive Descent Parsing</td>
</tr>
<tr>
<td>7</td>
<td><em>Bison</em>, a Parser Generator</td>
</tr>
<tr>
<td>8</td>
<td>Top-Down Parsing</td>
</tr>
<tr>
<td>9</td>
<td>Bottom-Up Parsing</td>
</tr>
</tbody>
</table>

Organisation of Lectures
Recommended books

   Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman

2. The C Programming Language
   Brian W. Kernighan and Dennis M. Ritchie

3. A Book on C
   Al Kelley and Ira Pohl
Recommended books

- The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie
- A Book on C Programming by Al Kelley and Ira Pohl
Don’t break the flow!

You view lecture notes \textit{in advance}

You do \textit{exercises}

I explain \textit{in lecture}