Lexical and Syntax Analysis

(of Programming Languages)

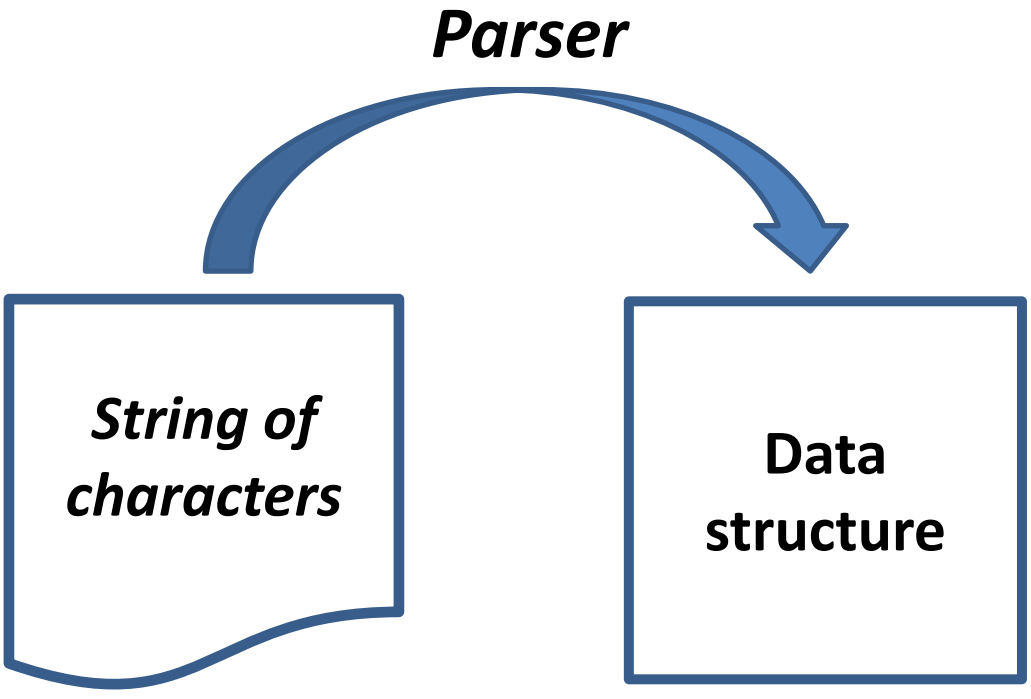
Introduction

Lexical and Syntax Analysis

(of Programming Languages)

Introduction

What is Parsing?

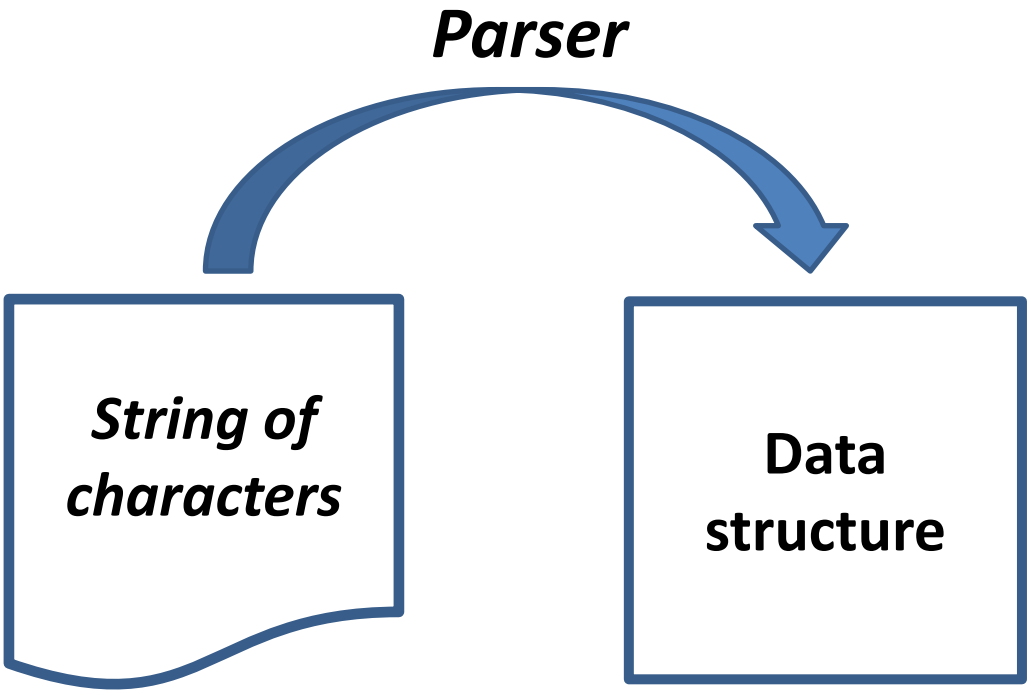


*Easy for **humans** to **write***

*Easy for **programs** to **process***

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

What is Parsing?

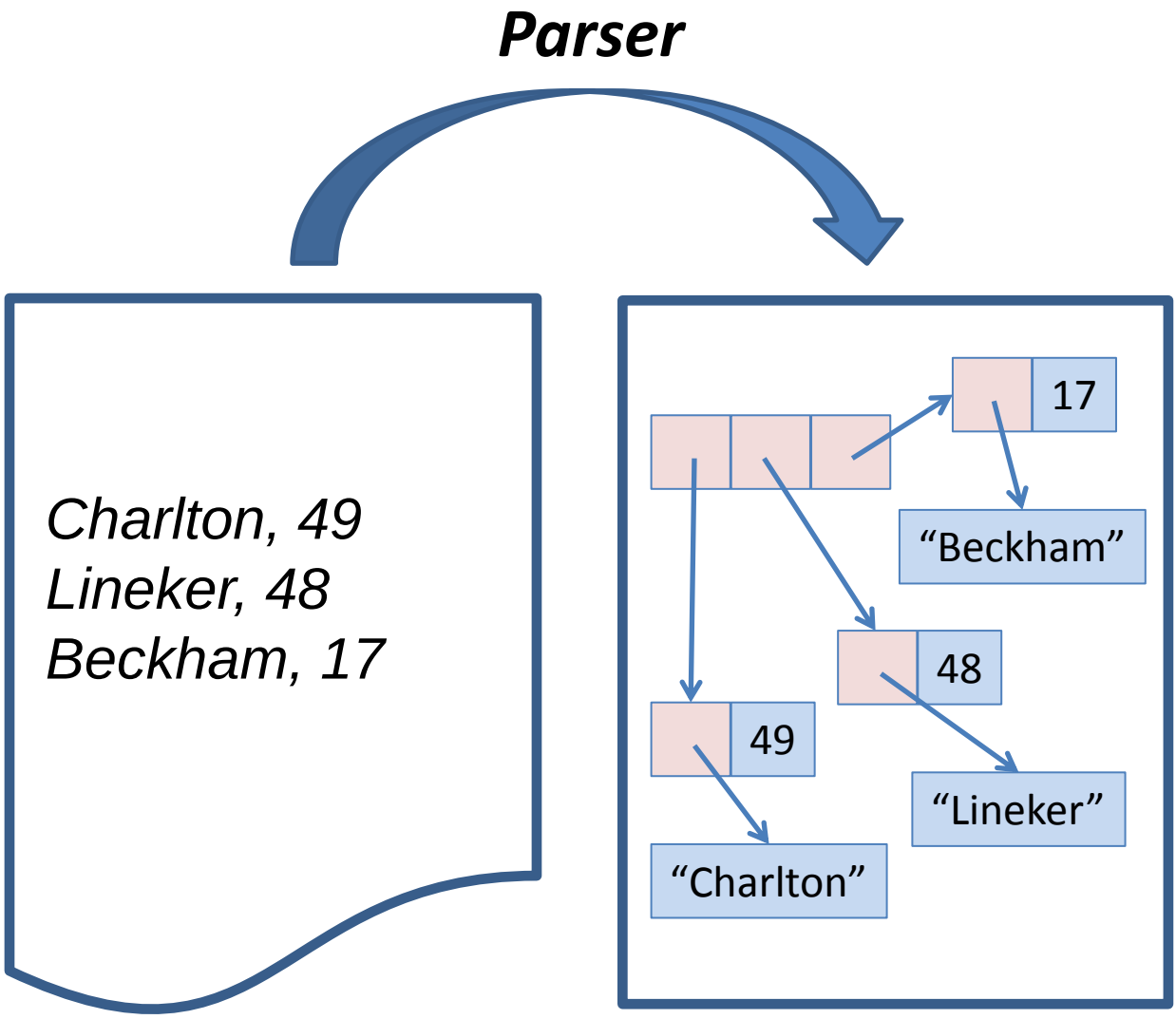


*Easy for **humans** to **write***

*Easy for **programs** to **process***

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

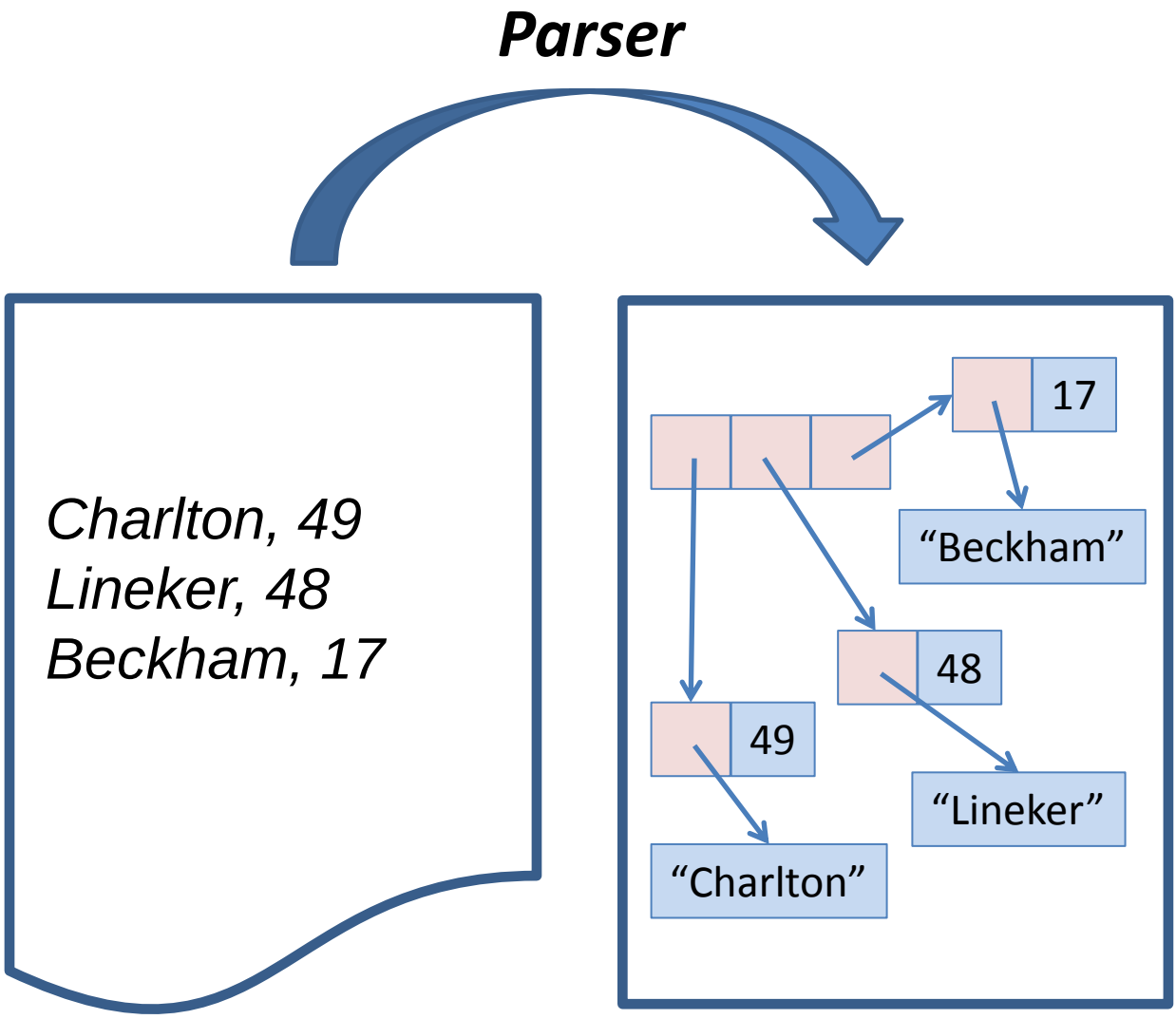
Example 1



CSV *(Comma Separated Value)*

Array of pairs

Example 1



CSV *(Comma Separated Value)*

Array of pairs

Data structure?

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

The type of a player:

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

The type of a squad of players:

```
typedef struct {
   Player* players;
   int size;
} Squad;
```

Data structure?

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

The type of a player:

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

The type of a squad of players:

```
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:

```
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:

```
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**.

```
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.

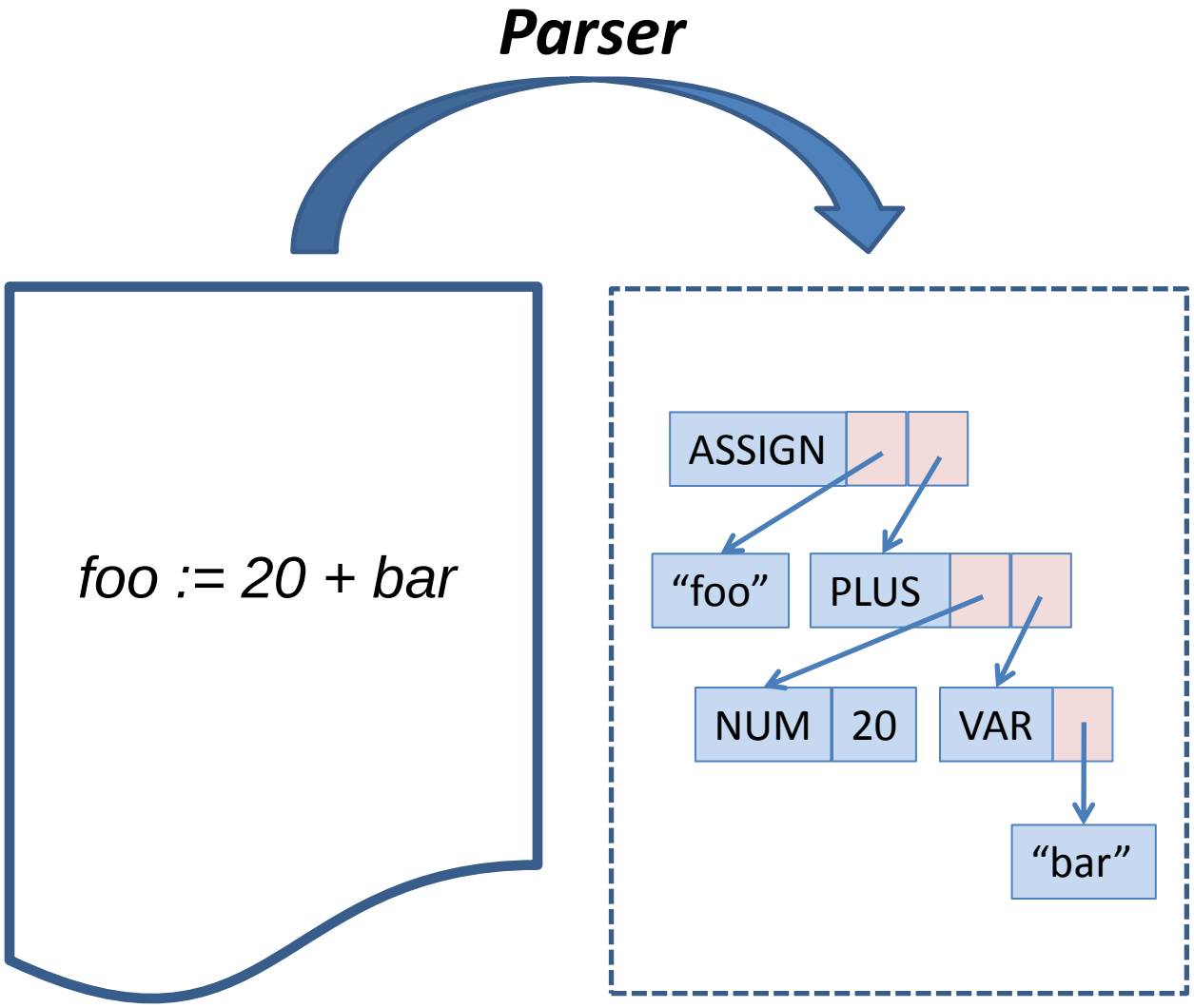
LSA of **PL**s

- 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 **PL**s

- 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

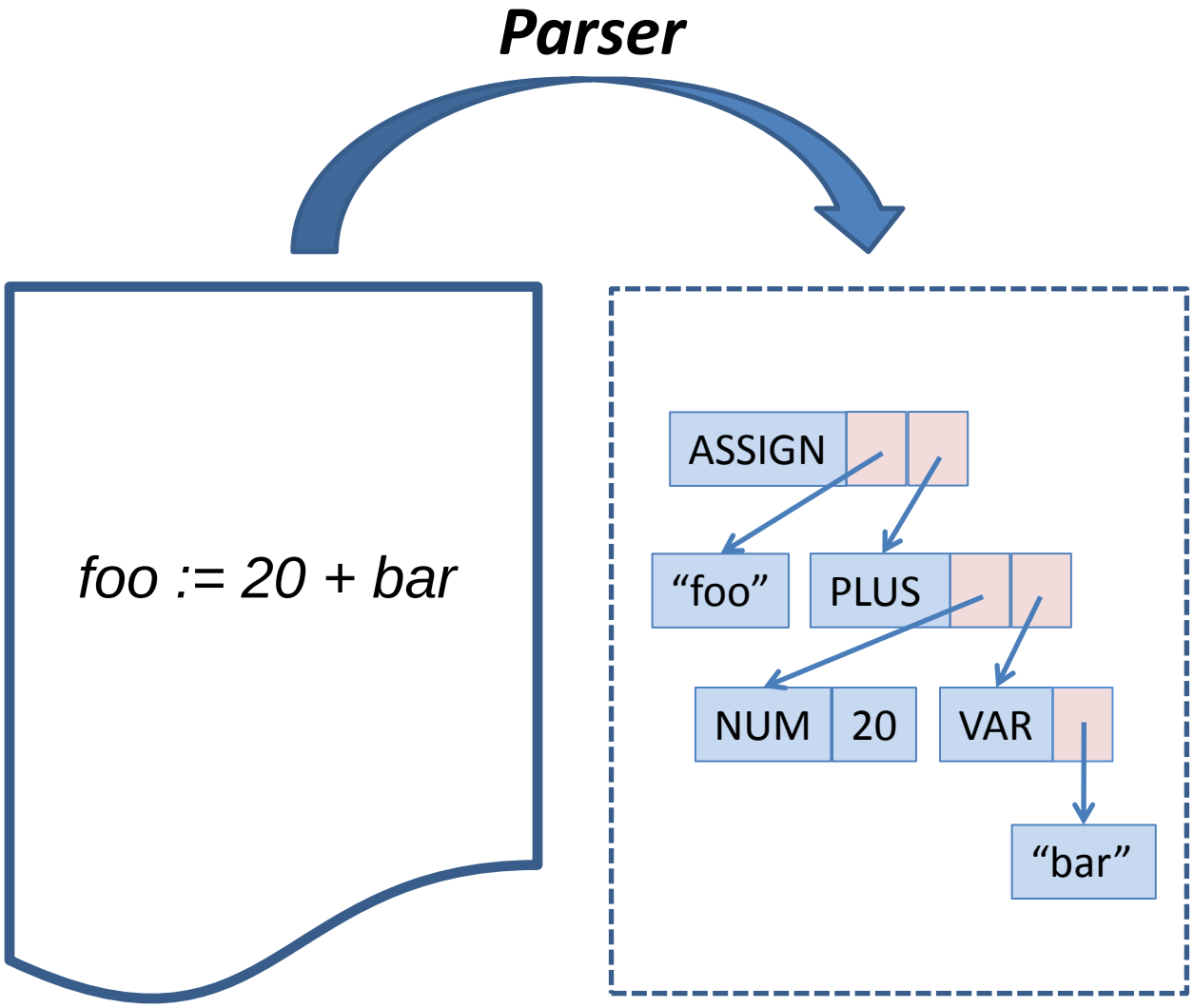


A pascal statement

An abstract syntax tree

We will return to this example later!

Example 2



A pascal statement

An abstract syntax tree

We will return to this example later!

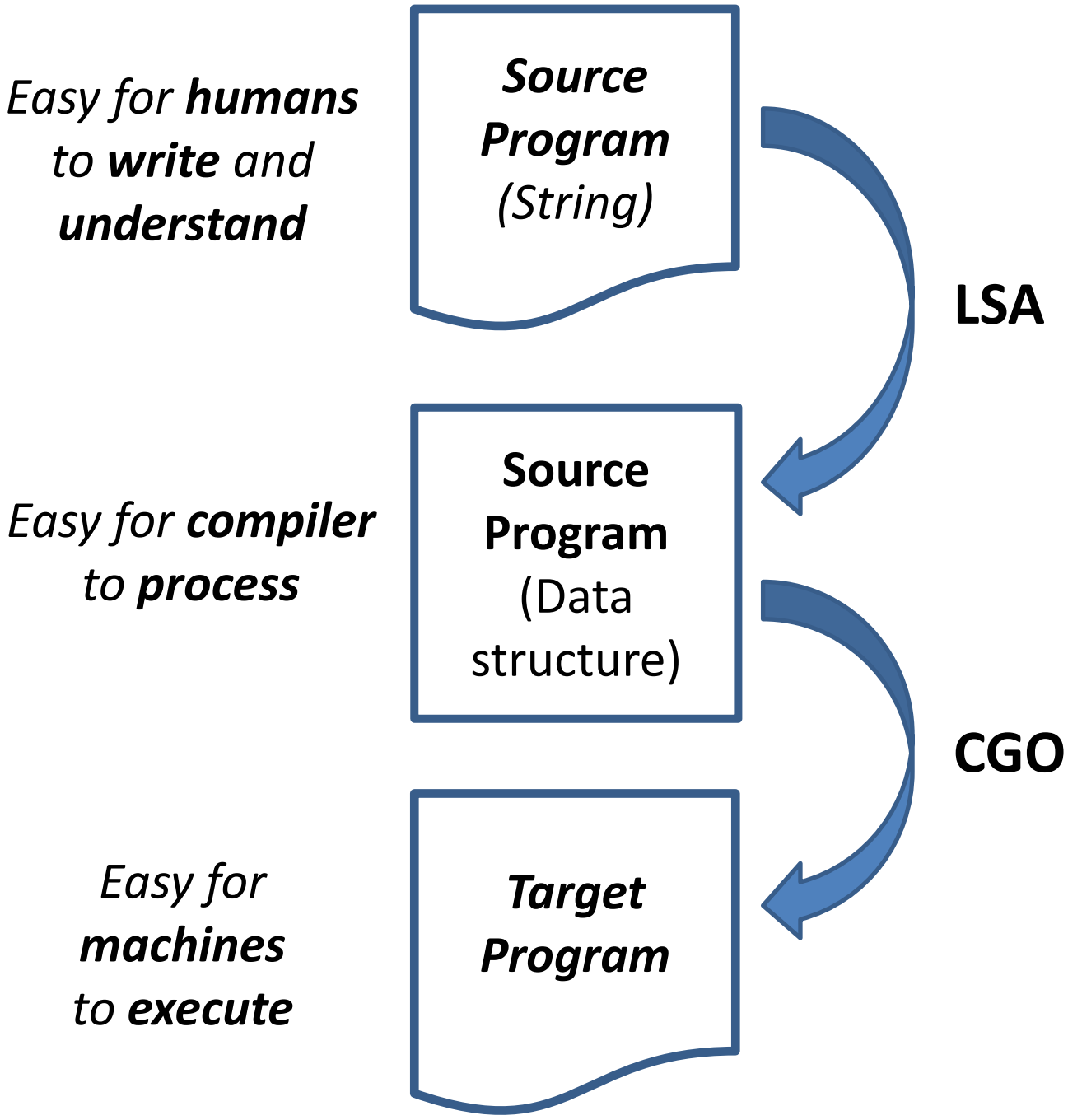
LSA & CGO

The connection between ***Lexical and Syntax Analysis*** (2nd year module) and ***Code Generation and Optimisation*** (3rd year module).

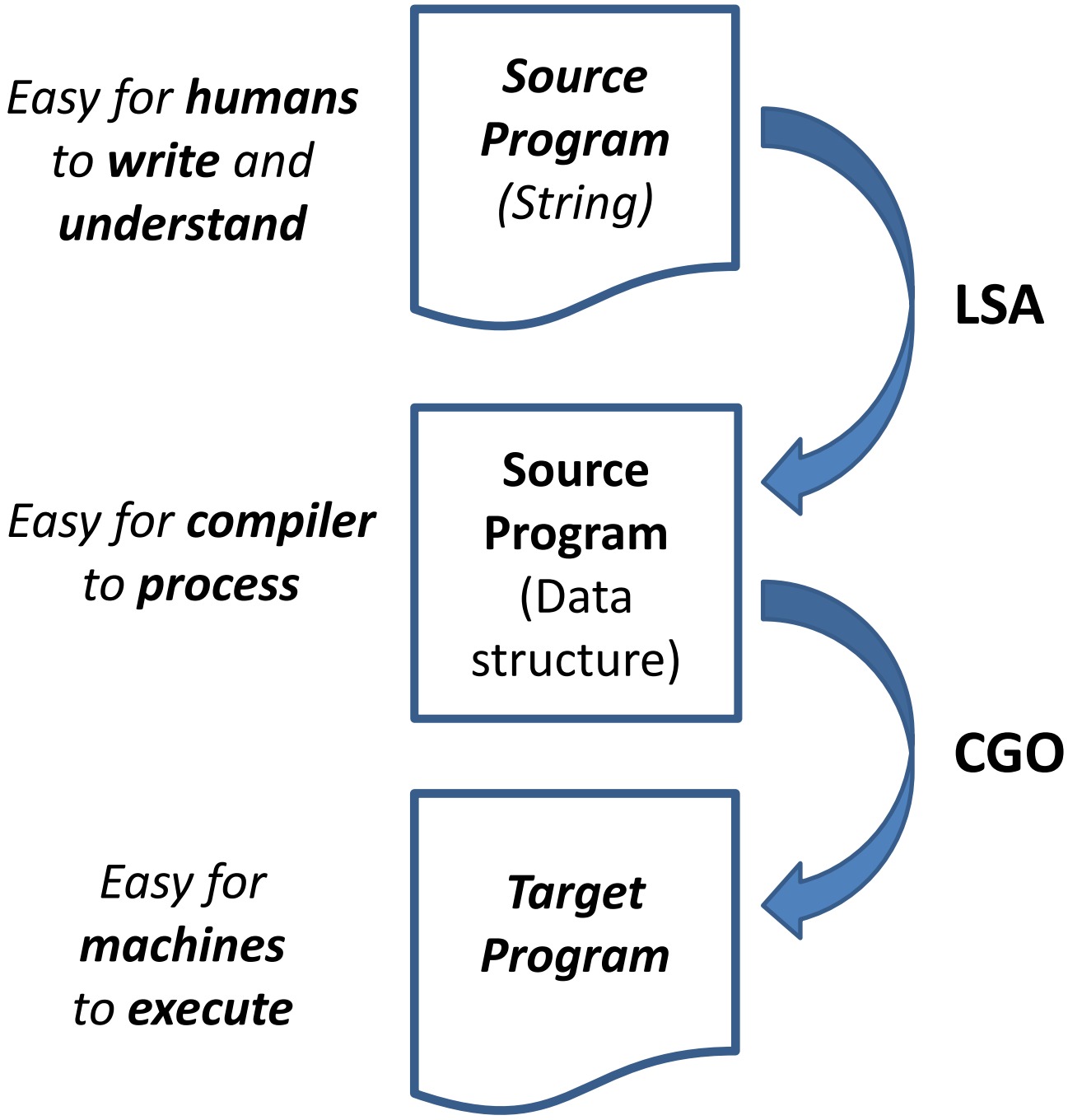
LSA & CGO

The connection between ***Lexical and Syntax Analysis*** (2nd year module) and ***Code Generation and Optimisation*** (3rd year module).

LSA & CGO


Easy for **humans** to **write** and **understand**
Source Program (String)
LSA
Easy for **compiler** to **process**
Source Program (Data structure)
CGO
Easy for **machines** to **execute**
Target Program

LSA & CGO


Easy for **humans** to **write** and **understand**
Source Program (String)
LSA
Easy for **compiler** to **process**
Source Program (Data structure)
CGO
Easy for **machines** to **execute**
Target Program

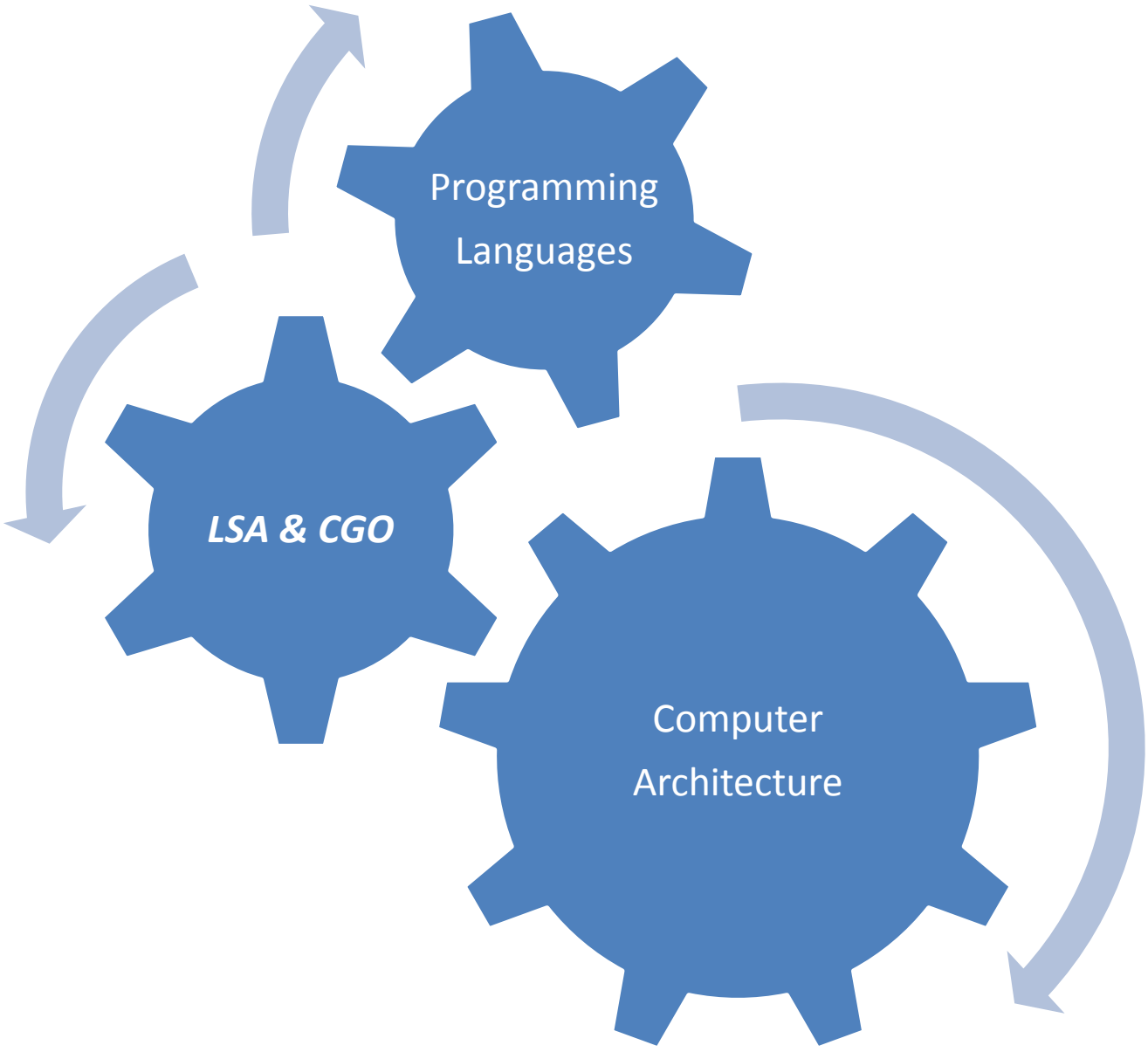
OTHER MOTIVATIONS

For studying parsing

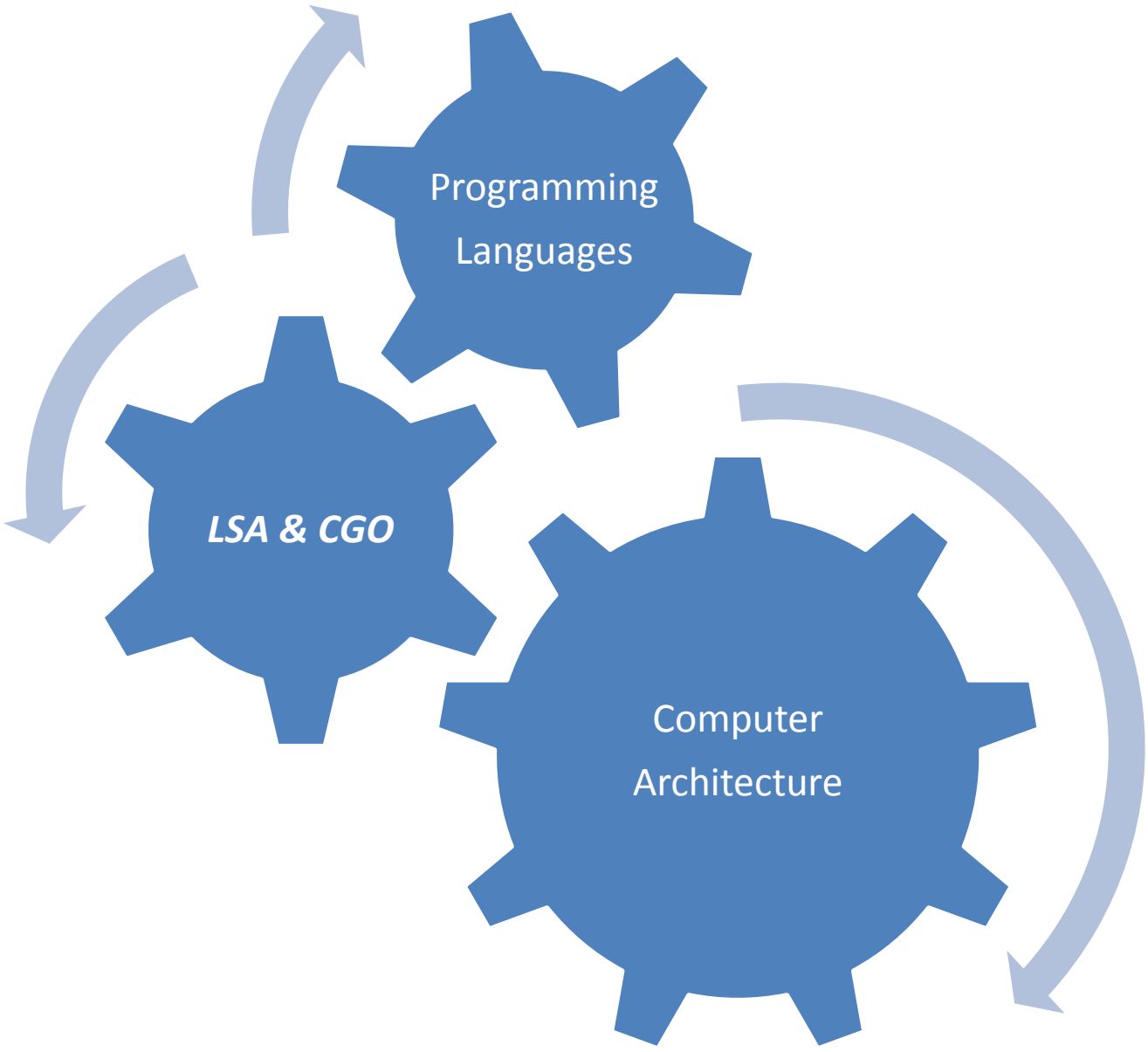
OTHER MOTIVATIONS

For studying parsing

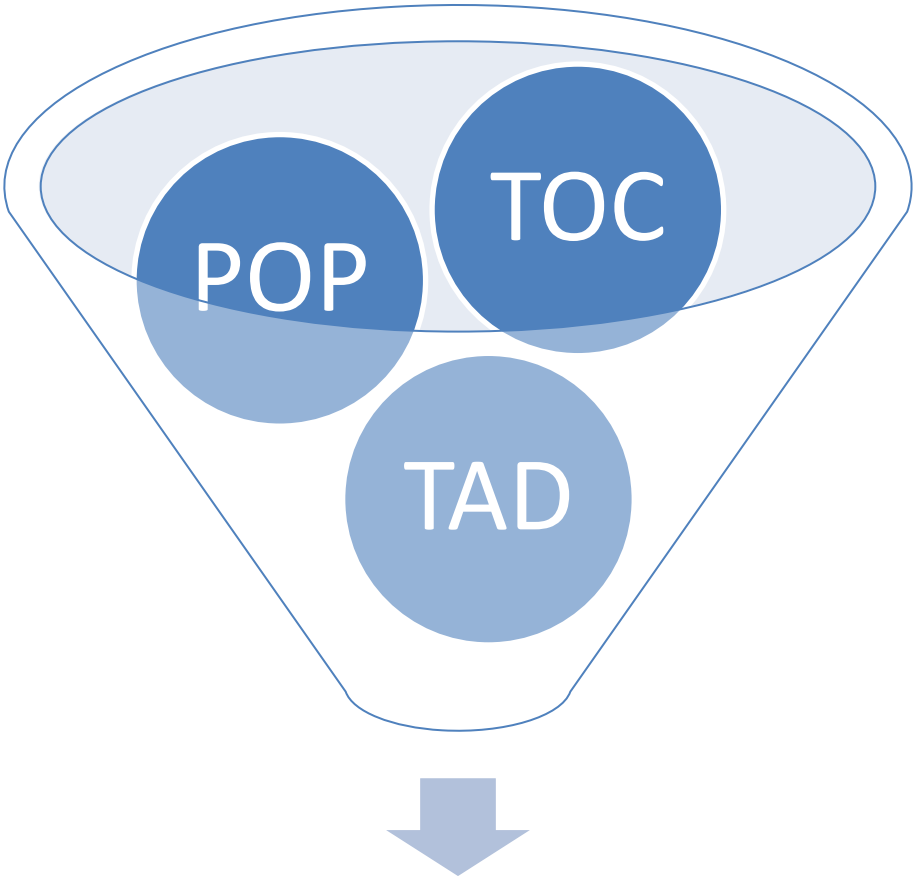
To understand *how computers work*



To understand *how computers work*

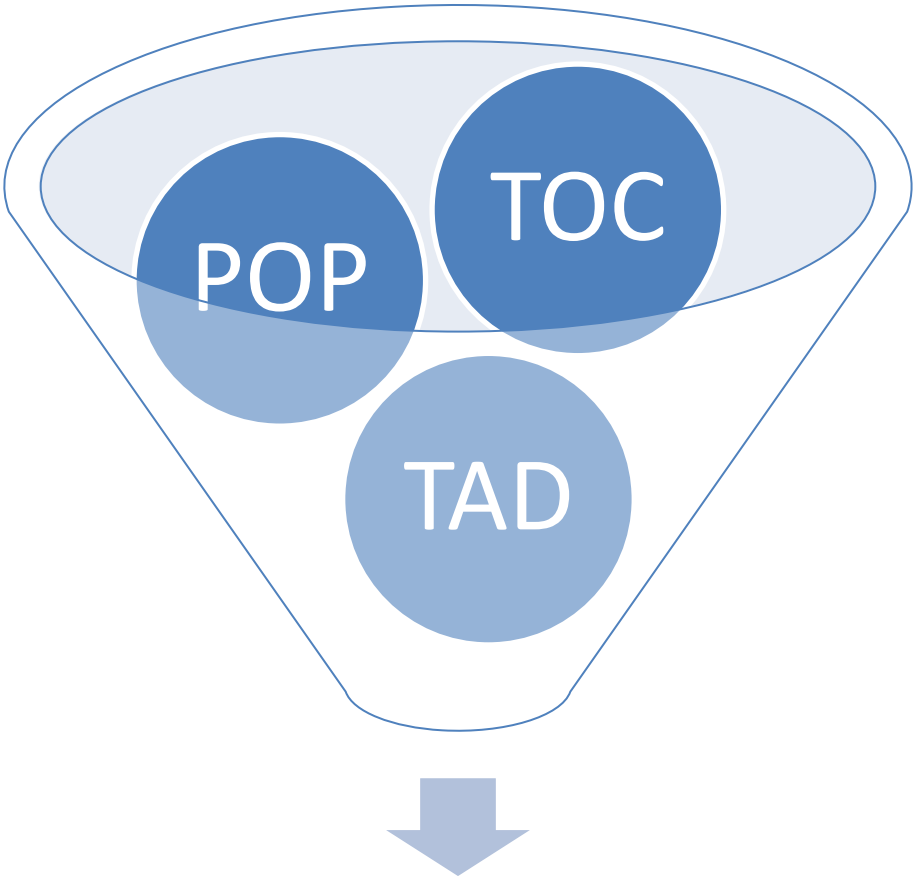


To practice *applying your knowledge*



LSA

To practice *applying your knowledge*



LSA

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.

Token identifier	denotes a
PLUS	+ operator
ASSIGN	:= operator
VAR	variable
NUM	number

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

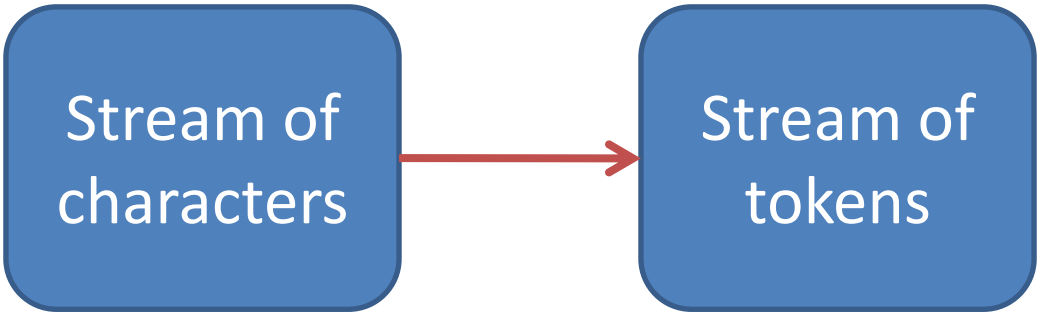
What is a token?

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

Token identifier	denotes a
PLUS	+ operator
ASSIGN	:= operator
VAR	variable
NUM	number

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

Lexical Analysis



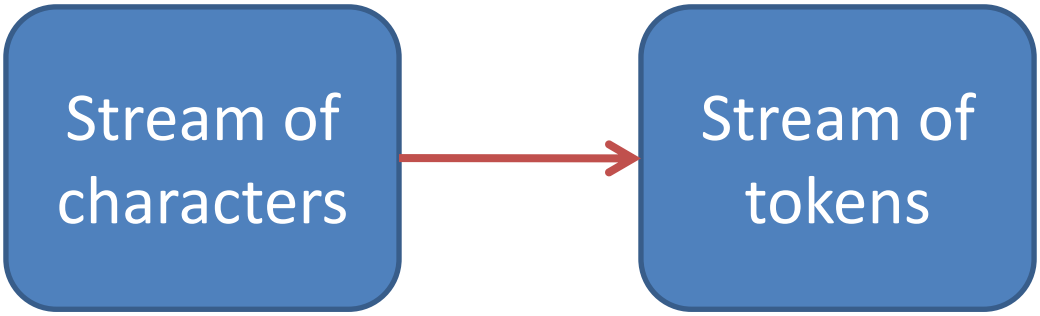
Example input:

```
foo := 20 + bar
```

Example output:

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

Lexical Analysis



Example input:

```
foo := 20 + bar
```

Example output:

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

Lexical Analysis

Lexemes are specified by **regular expressions**. For example:

digit	=	*0	...	9*
letter	=	*a	...	z*
number	=	*digit* · *digit**
variable	=	*letter* · (*letter* | *digit*)*

Example numbers:

1
4
43
634

Example variables:

x
foo
foo2
x1y20

Lexical Analysis

Lexemes are specified by **regular expressions**. For example:

digit	=	*0	...	9*
letter	=	*a	...	z*
number	=	*digit* · *digit**
variable	=	*letter* · (*letter* | *digit*)*

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:

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

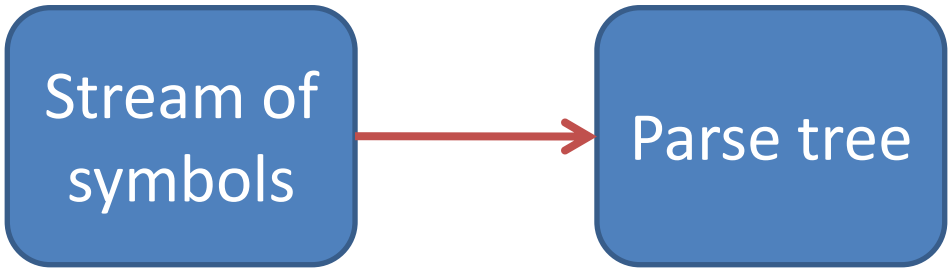
- The syntax of a language is usually specified by a **grammar**.
- Example:

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

stmt	→	*VAR(v) ASSIGN expr*

Where *v* represents any variable name and *n* any number.

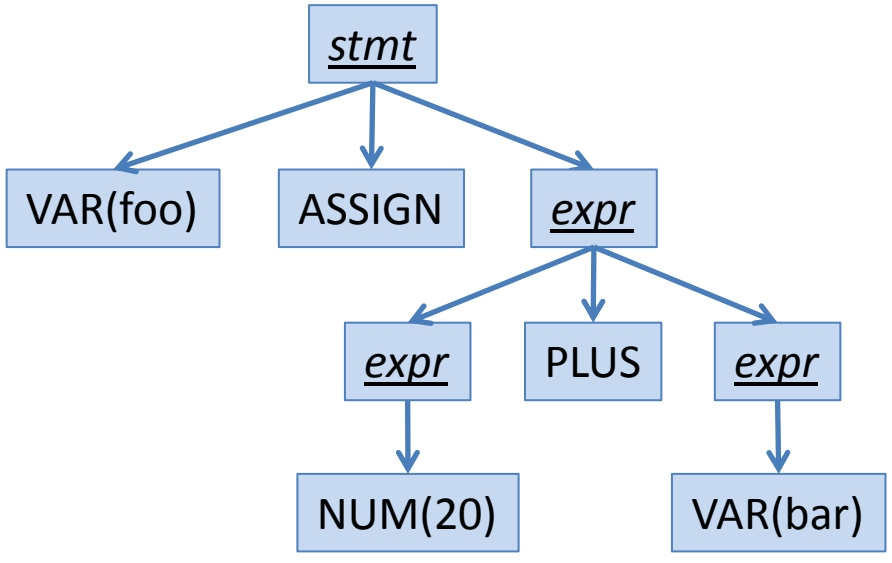
Syntax Analysis



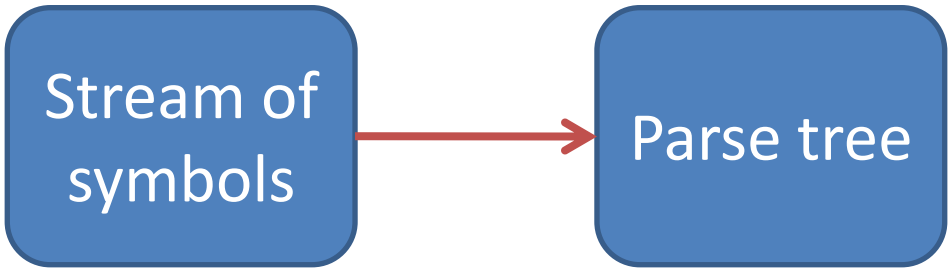
Example input:

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

Example output:



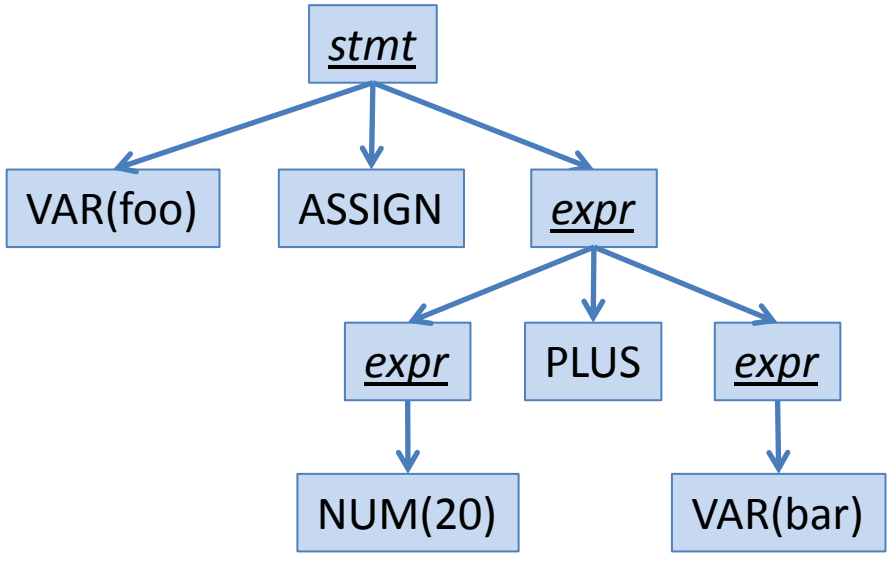
Syntax Analysis



Example input:

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

Example output:



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 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.

Practical	Topic
1	Lexical Analysis
2	*Flex*, a Lexical Analyser Generator
3	Recursive Descent Parsing
4	*Bison*, a Parser Generator

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.

Practical	Topic
1	Lexical Analysis
2	*Flex*, a Lexical Analyser Generator
3	Recursive Descent Parsing
4	*Bison*, a Parser Generator

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.

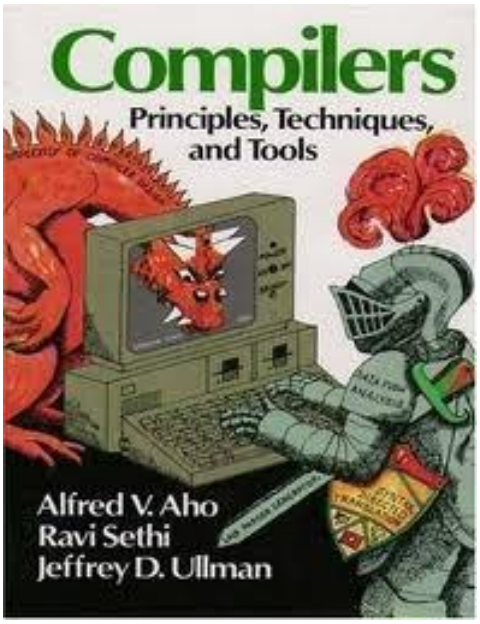
Chapter	Title
1	Introduction
2	Abstract Syntax
3	Lexical Analysis
4	*Flex*, a Lexical Analyser Generator
5	Grammars
6	Recursive Descent Parsing
7	*Bison*, a Parser Generator
8	Top-Down Parsing
9	Bottom-Up Parsing

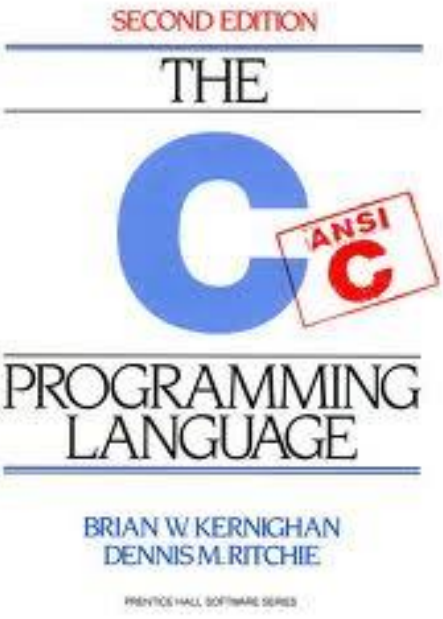
Lecture contents

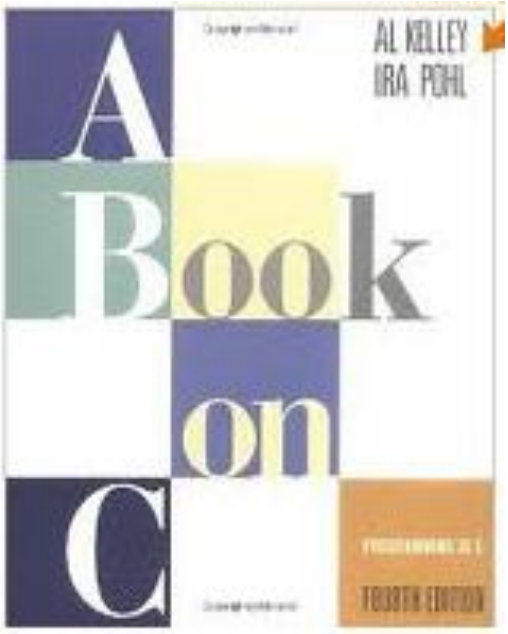
- 14 lectures, with notes arranged into 9 chapters.
- A single chapter may be covered in less than or more than one lecture.

Chapter	Title
1	Introduction
2	Abstract Syntax
3	Lexical Analysis
4	*Flex*, a Lexical Analyser Generator
5	Grammars
6	Recursive Descent Parsing
7	*Bison*, a Parser Generator
8	Top-Down Parsing
9	Bottom-Up Parsing

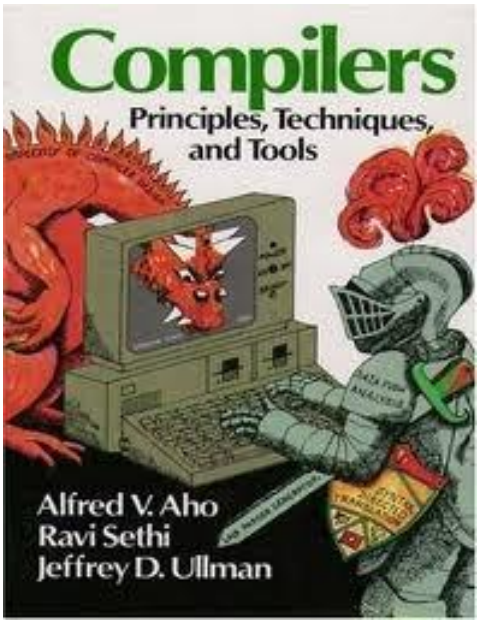
Recommended books

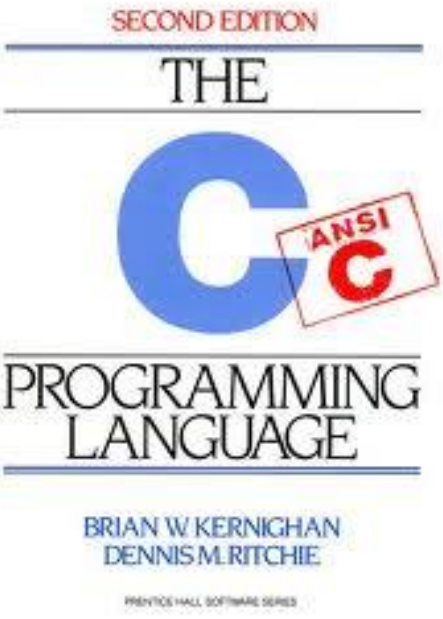


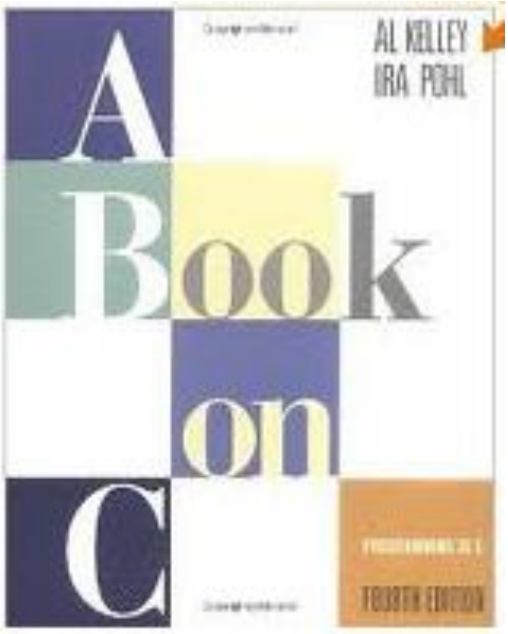




Recommended books







Don’t break **the flow**!