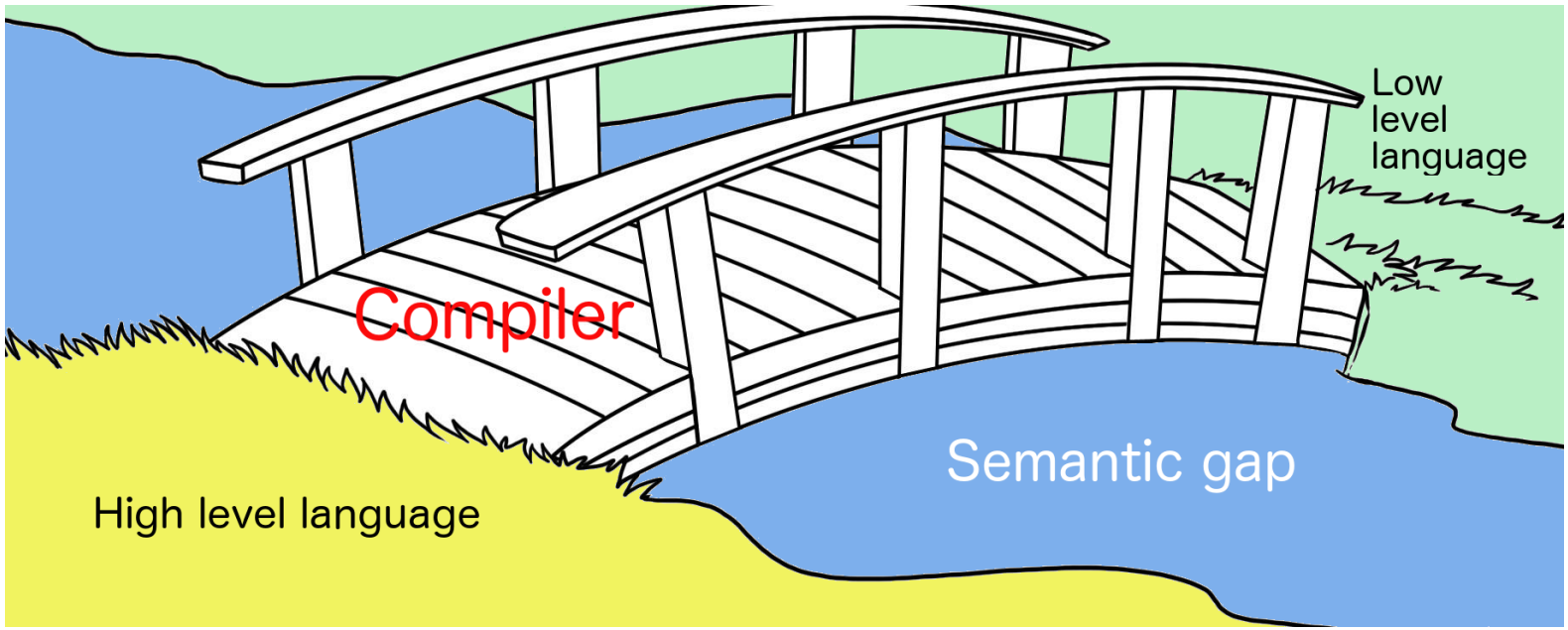
Compilers and computer architecture

2. From strings to ASTs: lexing and tokens

Ian Mackie

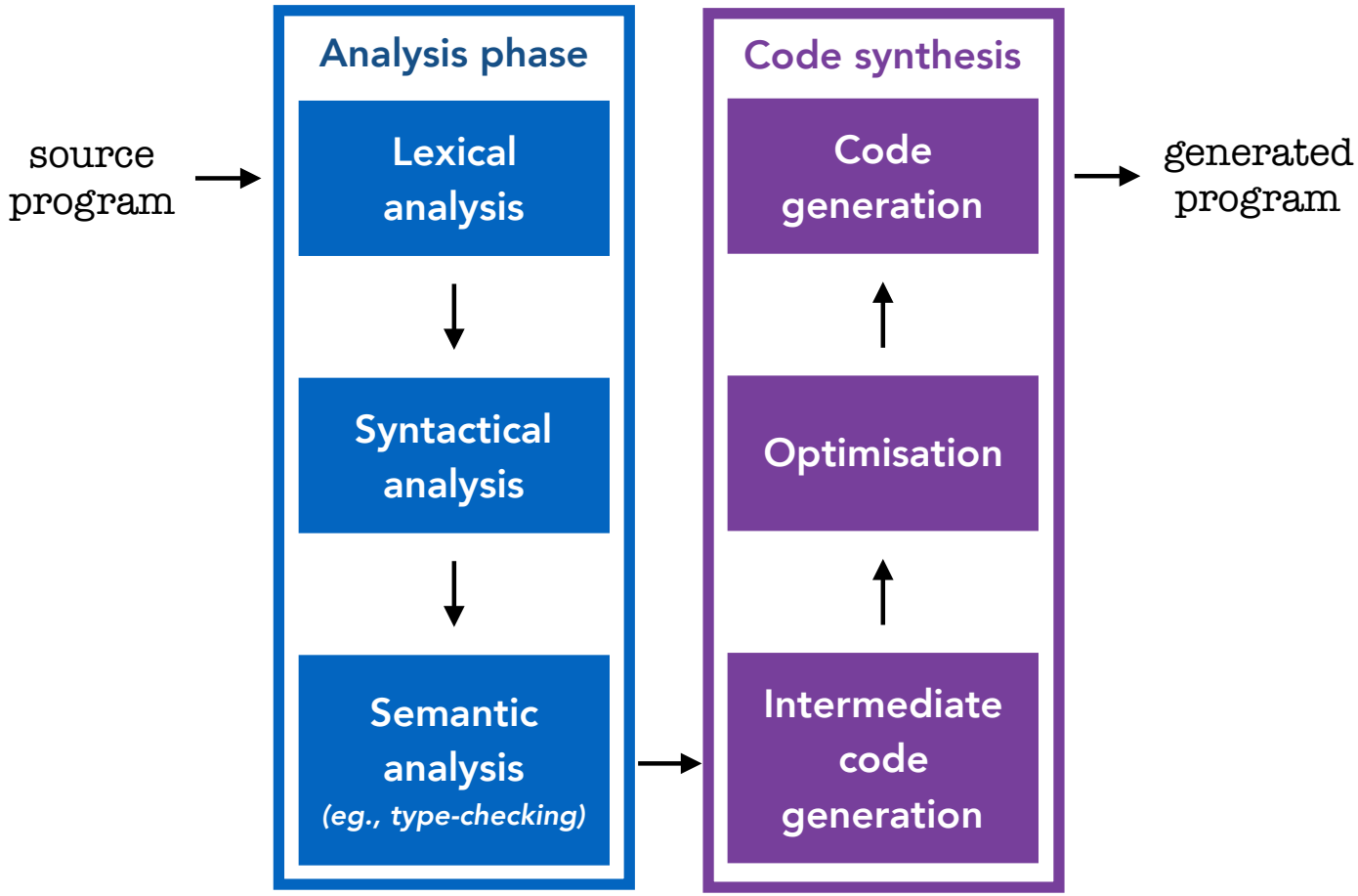
September 2022

Recall the function of compilers



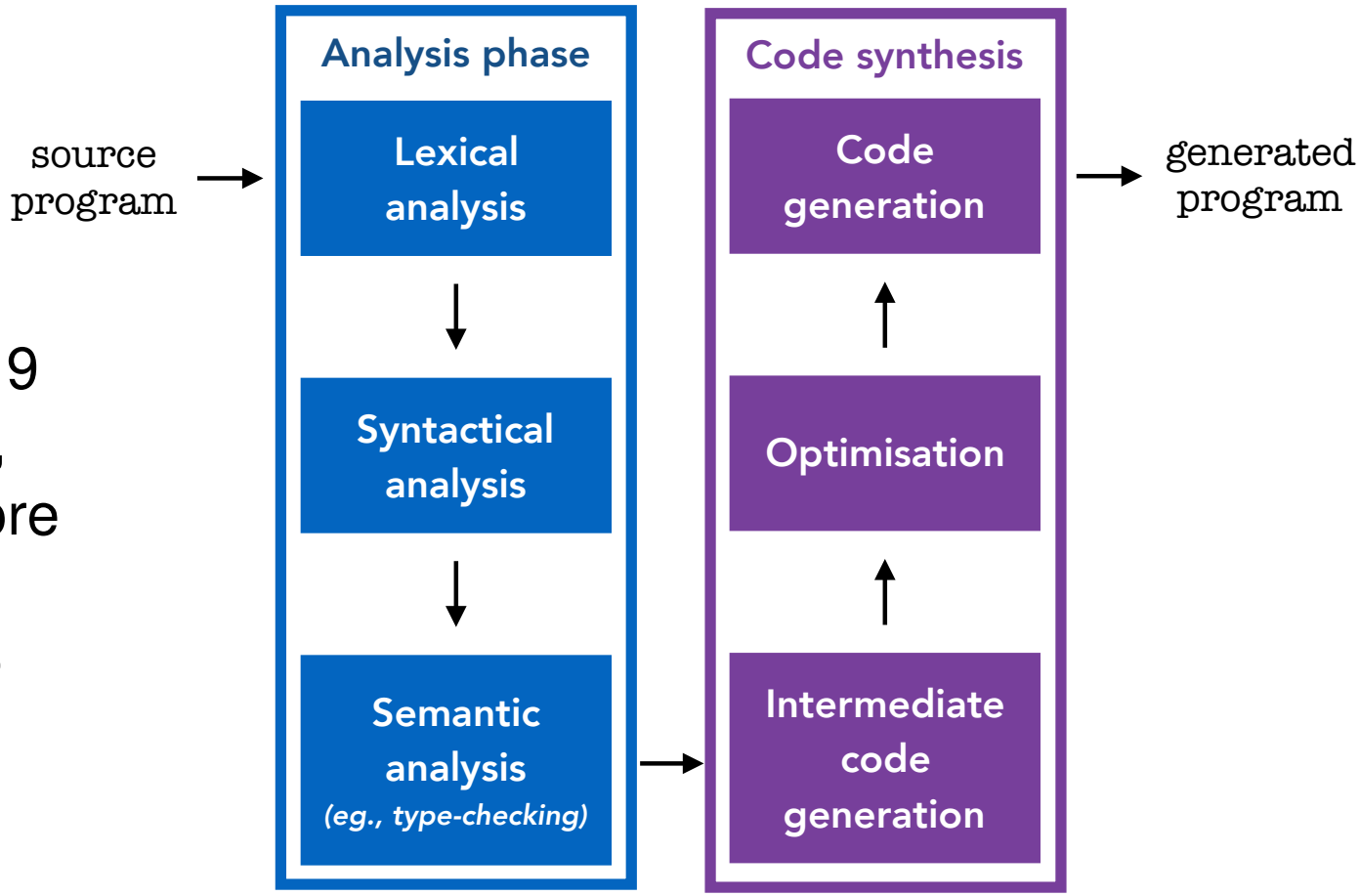
Plan for the next 9 weeks

Remember the shape of compilers:



Plan for the next 9 weeks

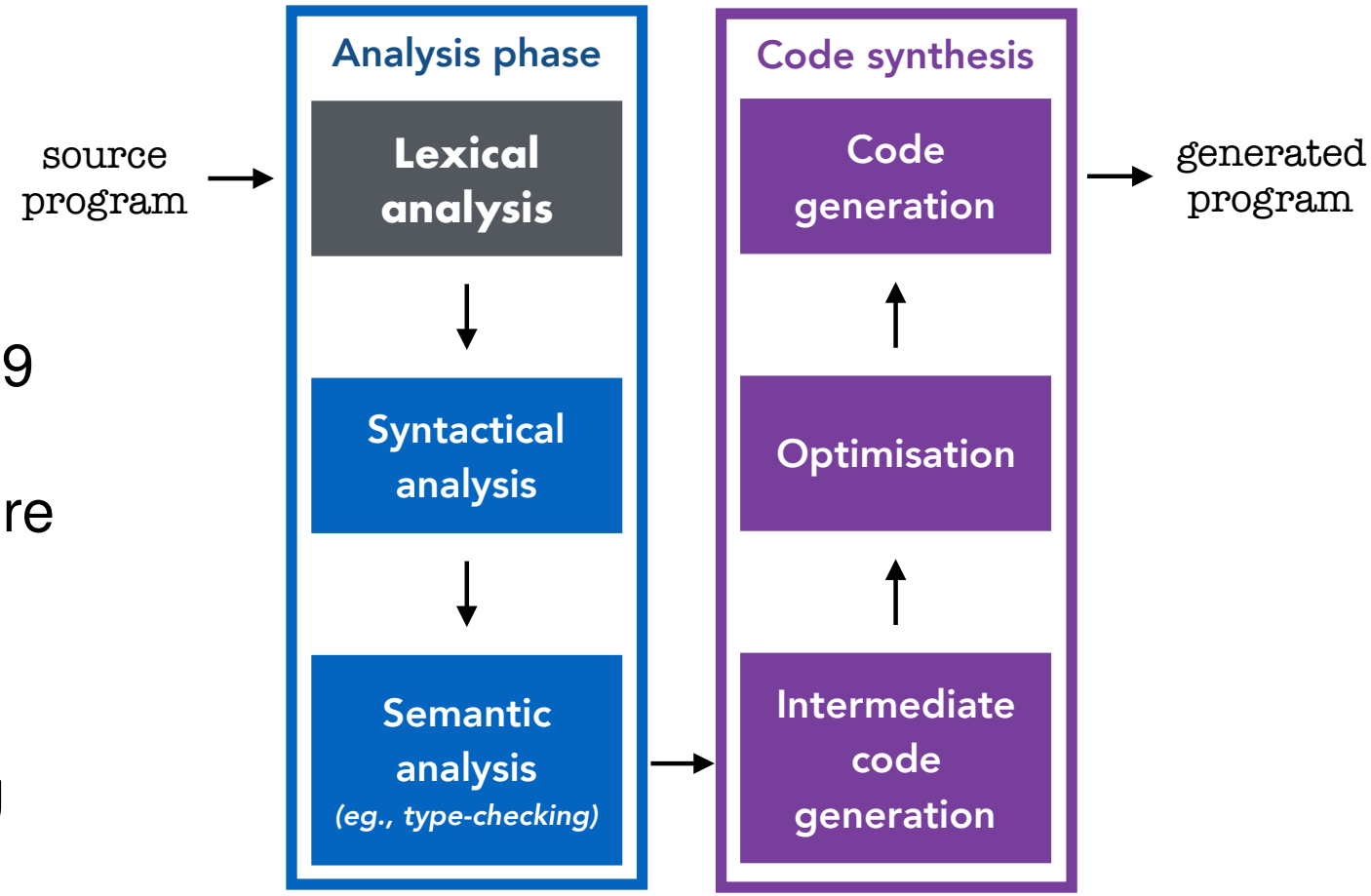
Remember the shape of compilers:



For the next 9 weeks or so, we will explore this pipeline step-by-step

Plan for the next 9 weeks

Remember the shape of compilers:



For the next 9 weeks or so, we will explore this pipeline step-by-step

— beginning with **lexing**.

From strings to ASTs

The purposes of the lexing and parsing phase is twofold.

- To convert the input from a string,
  to an abstract syntax tree (AST).

From strings to ASTs

The purposes of the lexing and parsing phase is twofold.

- To convert the input from a string,
  to an abstract syntax tree (AST).
  - strings: a representation that is (more or less)
    convenient for humans

From strings to ASTs

The purposes of the lexing and parsing phase is twofold.

- To convert the input from a string,
  to an abstract syntax tree (AST).
  - strings: a representation that is (more or less)
    convenient for humans
  - AST: a representation that is convenient
    for type-checking and code generation.

From strings to ASTs

The purposes of the lexing and parsing phase is twofold.

- To convert the input from a string,
  to an abstract syntax tree (AST).
  - strings: a representation that is (more or less)
    convenient for humans
  - AST: a representation that is convenient
    for type-checking and code generation.

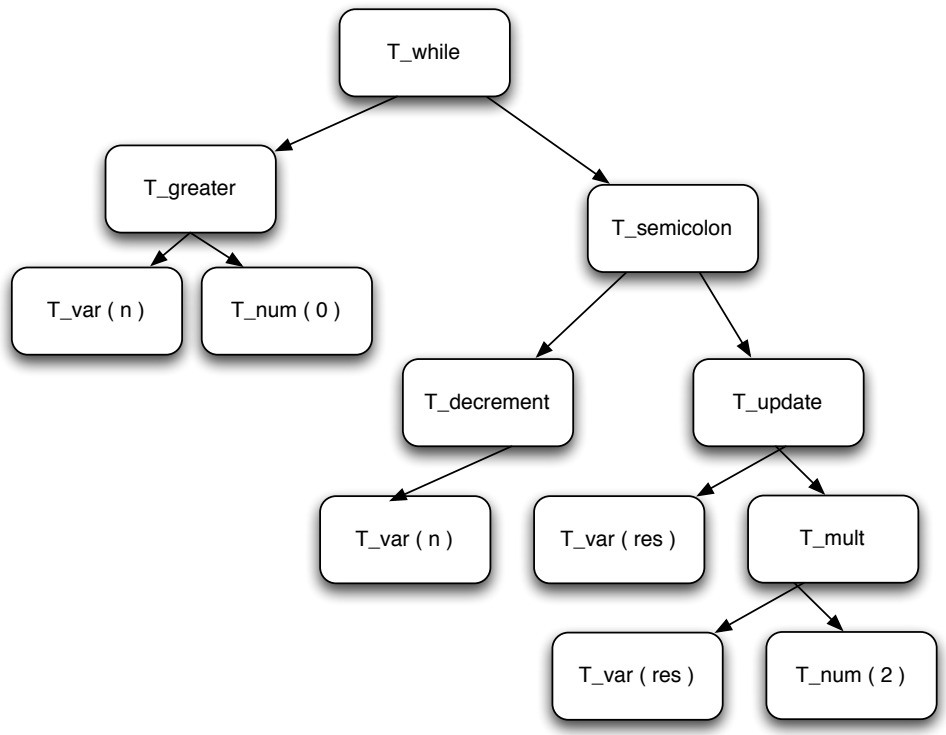
- To check the syntactic correctness of the program.

From strings to ASTs

We will define what ASTs are in detail, later.
For now, a picture says more than 1000 words.

We want to go from the representation on the left,
to that on the right:

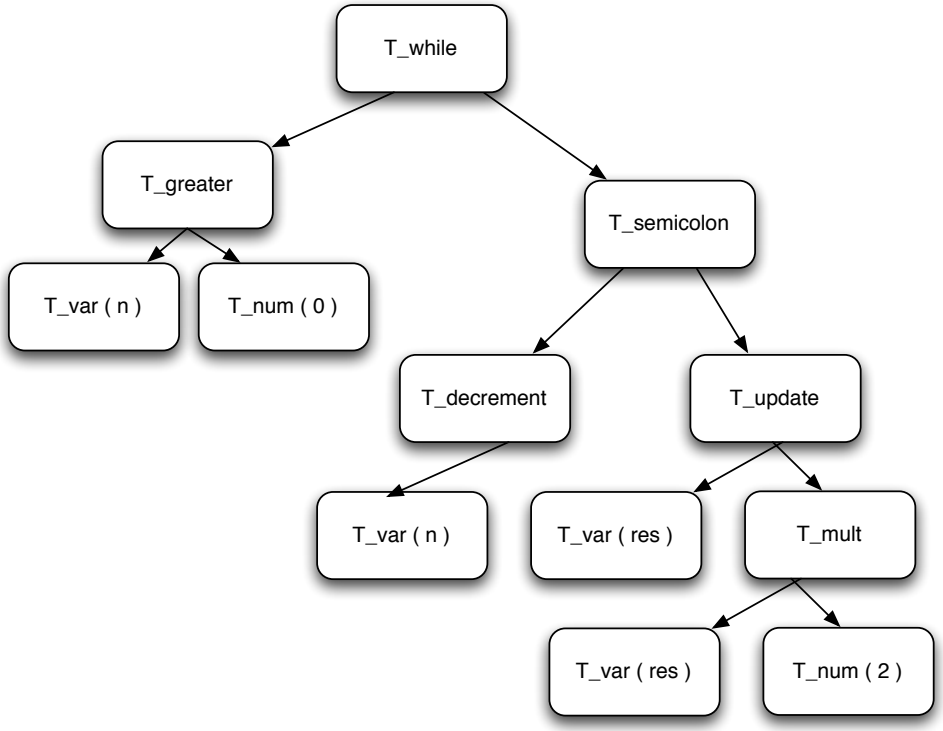
```
while (n>0) {
  n--;
  res *= 2;
}
```



From strings to ASTs

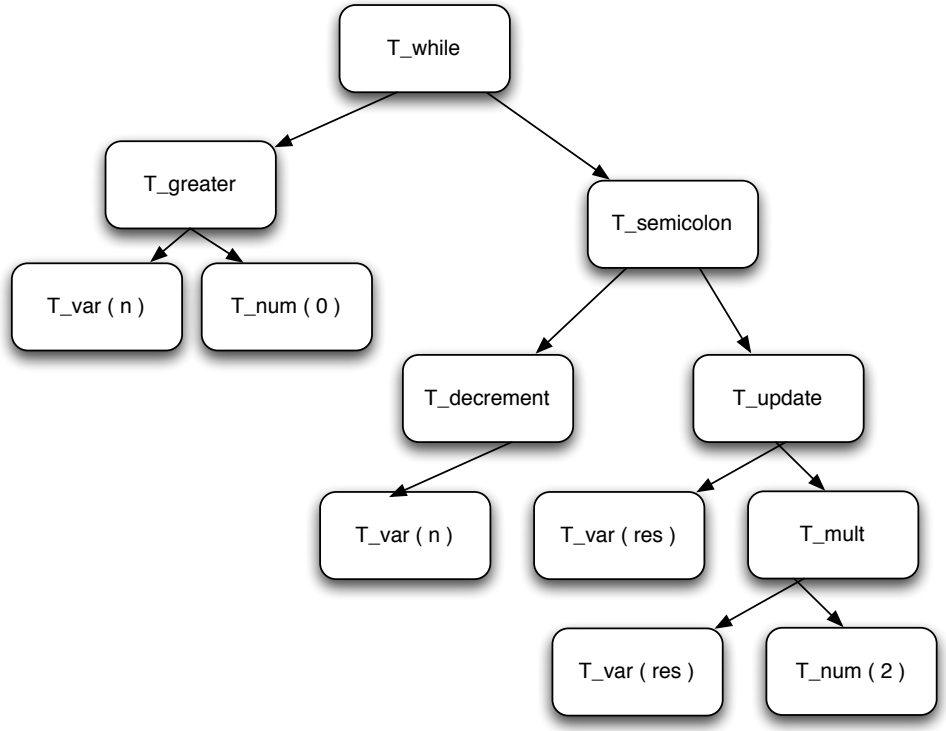
```
while (n>0) {
  n--;
  res *= 2;
}
```

Why?



From strings to ASTs

```
while (n>0) {
  n--;
  res *= 2;
}
```



Why?

To make type-checking and code synthesis — which depend on the **structure** of the program — easier and faster.

e.g., to generate code for `if C then Q else R`, we need to generate code for `C`, `Q`, and `R`.

- This is difficult, directly from strings.
- ASTs use pointers (references) to point to program components; CPUs manipulate pointers very efficiently.

From strings to ASTs

Most compilers use two steps:

- Lexical analysis (lexing);
- Syntactical analysis (parsing).

From strings to ASTs

Most compilers use two steps:

- Lexical analysis (lexing);
- Syntactical analysis (parsing).

This involves some language theory and automata, some of which you have already seen last year in one of the introductory courses. You will also learn about tools for generating lexers and parsers.

Lexical analysis (lexing)

The purpose of lexing is to prepare the input for the parser.
This involves related tasks:

- Removing irrelevant detail.
  (e.g. whitespace, tab-stops, new-lines)

- Split the string into primitive parts of language
  ('atoms' of the program), called **tokens**.

Lexical analysis (lexing)

Why is it good to use tokens?

Lexical analysis (lexing)

Why is it good to use tokens?

- For syntax analysis, it's **not** important if a variable is called `n` or `numUsers`, or a function is `toInt` vs. `toBox`, or a constant is `3` vs. `4`, etc.
  - For instance: in most languages, the code

    ```
    if n == 3 then x := 0 else x := 1
    ```

    will be **syntactically** correct exactly when

    ```
    if numUsers == 42 then y := 0 else z := 171
    ```

    is syntactically correct.

Lexical analysis (lexing)

Why is it good to use tokens?

- For syntax analysis, it's **not** important if a variable is called `n` or `numUsers`, or a function is `toInt` vs. `toBox`, or a constant is `3` vs. `4`, etc.
  - For instance: in most languages, the code

    ```
    if n == 3 then x := 0 else x := 1
    ```

    will be **syntactically** correct exactly when

    ```
    if numUsers == 42 then y := 0 else z := 171
    ```

    is syntactically correct.

- The precise choices of variable names, etc. are important for type-checking and for code generation — but we keep that information to check **later**.

Lexical analysis (lexing)

Another reason why using tokens is useful?

Lexical analysis (lexing)

Another reason why using tokens is useful?

- **Modularisation:** For syntax analysis, the actual keywords, symbols, etc. are **not** important — the structure is.
  - The conditional represented by

    ```
    if (n == 3) x = 0; else x = 1;
    ```

    in one language, is equivalent to a conditional

    ```
    if n = 3 then x := 0 else x := 1;
    ```

    in another language (despite different symbols / delimiters).

Lexical analysis (lexing)

Another reason why using tokens is useful?

- **Modularisation:** For syntax analysis, the actual keywords, symbols, etc. are **not** important — the structure is.
  - The conditional represented by

    ```
    if (n == 3) x = 0; else x = 1;
    ```

    in one language, is equivalent to a conditional

    ```
    if n = 3 then x := 0 else x := 1;
    ```

    in another language (despite different symbols / delimiters).

The precise choice of keywords name is irrelevant in later stages, so we abstract it away using tokens — making the rest of the compiler easy to apply to different languages.

Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

Lexical analysis (lexing)

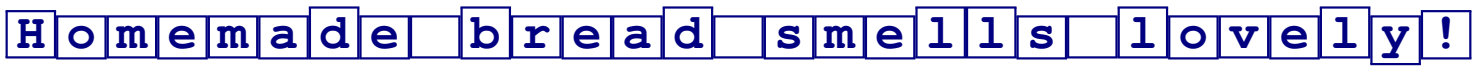
Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,

Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,

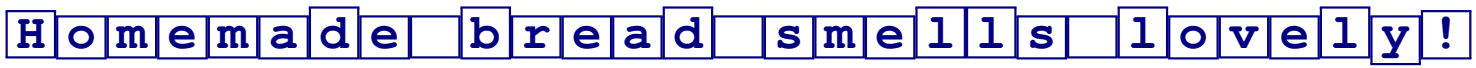


which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):

Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



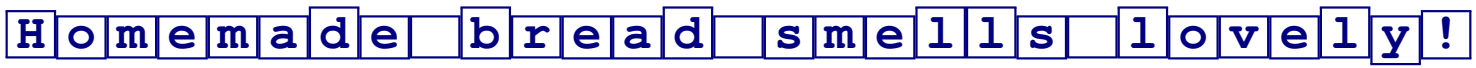
which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



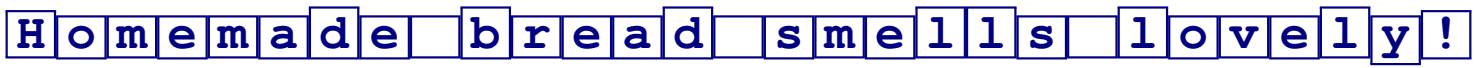
which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



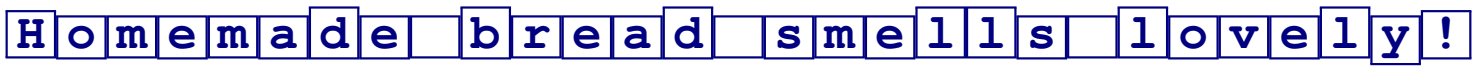
which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



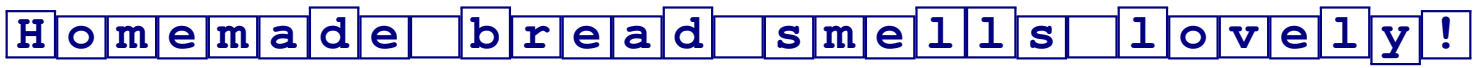
which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



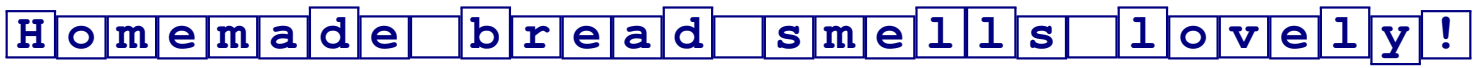
which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



Lexical analysis (lexing)

Let's look at an analogy — how **you** understand language.

A sentence in natural language (e.g., English) consists of a sequence of characters,



which you then interpret as 'words', with different syntactical roles (different roles as parts of speech):



If you succeed (e.g., you aren't stopped by spelling mistakes or unknown words), you then go on to assemble the words into sentences using just the list of the syntactical roles — [**adjective**, **noun**, **verb**, **adjective**, **punctuation**].

Lexical analysis (lexing)

To work out if an English sentence is syntactically correct, we can do the following:

- Perform lexical analysis: break the text into words (tokens), classified according to its function. (This is “lexing”.)

Lexical analysis (lexing)

To work out if an English sentence is syntactically correct, we can do the following:

- Perform lexical analysis: break the text into words (tokens), classified according to its function. (This is "lexing".)
- Check if the ordering of tokens is acceptable according to the grammar of English.

The first of these steps — **lexing** — is what takes us from the string `Homemade bread smells lovely!` to the list of tokens [**adjective**, **noun**, **verb**, **adjective**, **punctuation**].

Lexical analysis (lexing)

To work out if an English sentence is syntactically correct, we can do the following:

- Perform lexical analysis: break the text into words (tokens), classified according to its function. (This is "lexing".)
- Check if the ordering of tokens is acceptable according to the grammar of English.

The first of these steps — **lexing** — is what takes us from the string `Homemade bread smells lovely!` to the list of tokens [**adjective**, **noun**, **verb**, **adjective**, **punctuation**].

Question: How do we know what the words are?

Lexical analysis (lexing)

To work out if an English sentence is syntactically correct, we can do the following:

- Perform lexical analysis: break the text into words (tokens), classified according to its function. (This is "lexing".)
- Check if the ordering of tokens is acceptable according to the grammar of English.

The first of these steps — **lexing** — is what takes us from the string `Homemade bread smells lovely!` to the list of tokens [**adjective**, **noun**, **verb**, **adjective**, **punctuation**].

Question: How do we know what the words are?
Answer: We have a concept of a *boundary* (e.g., whitespace).

Lexical analysis (lexing)

To work out if an English sentence is syntactically correct, we can do the following:

- Perform lexical analysis: break the text into words (tokens), classified according to its function. (This is "lexing".)
- Check if the ordering of tokens is acceptable according to the grammar of English.

The first of these steps — **lexing** — is what takes us from the string `H o m e m a d e  b r e a d  s m e l l s  l o v e l y !` to the list of tokens [**adjective**, **noun**, **verb**, **adjective**, **punctuation**].

Question: How do we know what the words are?
Answer: We have a concept of a *boundary* (e.g., whitespace).

We do the same with computer languages.

Breaking down this process even further

Lexing / tokenisation *could* be broken down into two separate steps:

Breaking down this process even further

Lexing / tokenisation *could* be broken down into two separate steps:

- Start with **`String`**
- Split string on boundaries into **`List(String)`**
- Convert/classify the **`List(String)`** into **`List(Token)`**

Breaking down this process even further

Lexing / tokenisation *could* be broken down into two separate steps:

- Start with **`String`**
- Split string on boundaries into **`List(String)`**
- Convert/classify the **`List(String)`** into **`List(Token)`**

Example:

- **`String`**:
  `Homemade bread smells lovely!`
- **`List(String)`**:
  [`Homemade`, `bread`, `smells`, `lovely`, `!`]
- **`List(Token)`**:
  [**adjective**, **noun**, **verb**, **adjective**, **punctuation**]

Breaking down this process even further

Lexing / tokenisation *could* be broken down into two separate steps:

- Start with **`String`**
- Split string on boundaries into **`List(String)`**
- Convert/classify the **`List(String)`** into **`List(Token)`**

Example:

- **`String`**:
  `Homemade bread smells lovely!`
- **`List(String)`**:
  [`Homemade`, `bread`, `smells`, `lovely`, `!`]
- **`List(Token)`**:
  [**adjective**, **noun**, **verb**, **adjective**, **punctuation**]

In practise, those steps are often excuted in one pass.

Question: why bother with this?

Recall that our goal is twofold:

Question: why bother with this?

Recall that our goal is twofold:

- Produce an AST form the input string, because code generation works with ASTs.

Question: why bother with this?

Recall that our goal is twofold:

- Produce an AST form the input string, because code generation works with ASTs.
- Check syntactic correctness.

Question: why bother with this?

Recall that our goal is twofold:

- Produce an AST form the input string, because code generation works with ASTs.
- Check syntactic correctness.

Both are **much** easier using a list of tokens, rather than a string — and neither depend crucially on the precise names used, or the precise form of the keywords / symbols (e.g., `=` vs. `==`).

Question: why bother with this?

Recall that our goal is twofold:

- Produce an AST form the input string, because code generation works with ASTs.
- Check syntactic correctness.

Both are **much** easier using a list of tokens, rather than a string — and neither depend crucially on the precise names used, or the precise form of the keywords / symbols (e.g., `=` vs. `==`).

$\Longrightarrow$ tokenisation is a form of simplification (information hiding), shielding later stages from irrelevant information.

In summary, lexing allows the compiler to abstract away irrelevant information about the source code, simplifying the task of the later stages (particularly the next stage, parsing).

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`
- Integer: `0`, `1`, `17`, `-3`, ..., but not:
- Float: `2.0`, `3.14159`, `-16.993`, ...

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`
- Integer: `0`, `1`, `17`, `-3`, ..., but not:
- Float: `2.0`, `3.14159`, `-16.993`, ...
- BinOp: `+`, `*`, `&&`, `||`, `=`, `==`, `!=`, `:=`, ...
- LeftParen `(`, and RightParen `)`

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`
- Integer: `0`, `1`, `17`, `-3`, ..., but not:
- Float: `2.0`, `3.14159`, `-16.993`, ...
- BinOp: `+`, `*`, `&&`, `||`, `=`, `==`, `!=`, `:=`, ...
- LeftParen `(`, and RightParen `)`
- Ignored input: usually whitespace (*e.g.*, tab-stops, newlines)

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`
- Integer: `0`, `1`, `17`, `-3`, ..., but not:
- Float: `2.0`, `3.14159`, `-16.993`, ...
- BinOp: `+`, `*`, `&&`, `||`, `=`, `==`, `!=`, `:=`, ...
- LeftParen `(`, and RightParen `)`
- Ignored input: usually whitespace (*e.g.*, tab-stops, newlines)
- ...

Lexical analysis (lexing)

When designing a programming language or a compiler, we must decide what the basic parts of programs are, and what the grammatical structure of the language is.

One example:

- Keywords: e.g., `IF`, `ELSE`, `WHILE`, ..., but not `GIRAFFE`
- Ident: `x`, `i`, `username`, ..., but not `!_++17`
- Integer: `0`, `1`, `17`, `-3`, ..., but not:
- Float: `2.0`, `3.14159`, `-16.993`, ...
- BinOp: `+`, `*`, `&&`, `||`, `=`, `==`, `!=`, `:=`, ...
- LeftParen `(`, and RightParen `)`
- Ignored input: usually whitespace (*e.g.*, tab-stops, newlines)
- ...

So the tokens are: `IF`, `ELSE`, ..., Ident, Integer, Float, BinOp, LeftParen, RightParen, ...

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword,

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen,

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**),

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**),

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**), Integer(**1**),

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**), Integer(**1**), RightParen,

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**), Integer(**1**), RightParen, **THEN** keyword,

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**),
Integer(**1**), RightParen, **THEN** keyword, Ident(**z**),

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**), Integer(**1**), RightParen, **THEN** keyword, Ident(**z**), BinOp(**:=**),

Example

Let's look at the string

```
\t IF ( x > 1 \n) \t\t\n THEN \tz := -3.01 \n
```

If we use whitespace characters to delimit tokens but otherwise ignore them, this gives rise to the following token list:

IF keyword, LeftParen, Ident(**x**), BinOp(**>**), Integer(**1**), RightParen, **THEN** keyword, Ident(**z**), BinOp(**:=**), Float(**-3.01**)

Lexical analysis (lexing)
Ingredients

To lex strings we need to decide on the following ingredients.

- Tokens (representing sets of strings).

Lexical analysis (lexing)
Ingredients

To lex strings we need to decide on the following ingredients.

- Tokens (representing sets of strings).
- **Inhabitation** of tokens: i.e. decide how strings are classified by which tokens.

Lexical analysis (lexing)
Ingredients

To lex strings we need to decide on the following ingredients.

- Tokens (representing sets of strings).
- **Inhabitation** of tokens: i.e. decide how strings are classified by which tokens.
- An algorithm that inputs a string, and outputs a token list.

Lexical analysis (lexing)
Ingredients

To lex strings we need to decide on the following ingredients.

- Tokens (representing sets of strings).
- **Inhabitation** of tokens: i.e. decide how strings are classified by which tokens.
- An algorithm that inputs a string, and outputs a token list.
- What to do if we encounter an input that isn't well-formed — a string that cannot be broken down into a list of tokens (e.g., having invalid characters).

Example: Tokens and their inhabitation

Suppose that we have a language in which

- The token class for Identifiers is inhabited by any non-empty string
  - letters,
  - digits,
  - and "_" (underscore),

  . . . and which starts with a letter.
- The token class Integers is inhabited by non-empty strings of digits, but not starting with `0`.
- Keywords: `ELSE`, `IF`, `NEW`, . . . each is its own token class.
- Whitespace: a non-empty string of blanks, newlines and tab-stops. **Not** inhabiting any token classes, since it is irrelevant to rest of compilation.

Example: Tokens and their inhabitation

Suppose that we have a language in which

- The token class for Identifiers is inhabited by any non-empty string
  - letters,
  - digits,
  - and "_" (underscore),

  . . . and which starts with a letter.
- The token class Integers is inhabited by non-empty strings of digits, but not starting with `0`.
- Keywords: `ELSE`, `IF`, `NEW`, . . . each is its own token class.
- Whitespace: a non-empty string of blanks, newlines and tab-stops. **Not** inhabiting any token classes, since it is irrelevant to rest of compilation.

Can you see the possible problems with this?

Tokens and their inhabitation

Some strings can be described by more than one token, e.g., as described `THEN` is an identifier and a keyword.

Tokens and their inhabitation

Some strings can be described by more than one token, e.g., as described `THEN` is an identifier and a keyword.

We must have a mechanism to deal with this —
e.g., decide that if a string is both a keyword and an identifier, it should be classified as a keyword. In other words, keywords have **priority** over identifiers.

Tokens and their inhabitation

Some strings can be described by more than one token, e.g., as described `THEN` is an identifier and a keyword.

We must have a mechanism to deal with this —
e.g., decide that if a string is both a keyword and an identifier, it should be classified as a keyword. In other words, keywords have **priority** over identifiers.

— Aside: Some old programming languages let keywords be identifiers. For example in PL/1 the following is valid (or something close to it):

```
if if = then ( if, else ) then else=1 else else=3
```

Allowing identifiers also to be keywords is rarely useful, so most modern programming languages prohibit it.

Tokens and their inhabitation

There is a second problem!

What about the string `IFTHEN`?

Tokens and their inhabitation

There is a second problem!

What about the string **`IFTHEN`**? Should it be the tokens **`IF`** followed by **`THEN`**, or a single token standing for the identifier **`IFTHEN`**?

Tokens and their inhabitation

There is a second problem!

What about the string `IFTHEN`? Should it be the tokens `IF` followed by `THEN`, or a single token standing for the identifier `IFTHEN`?

Usual answer: use **longest match**.

Tokens and their inhabitation

There is a second problem!

What about the string `IFTHEN`? Should it be the tokens `IF` followed by `THEN`, or a single token standing for the identifier `IFTHEN`?

Usual answer: use **longest match**.

So: `IFTHEN` is classed as an identifier: while it contains `IF` in its first two characters (i.e., as a *prefix*), we can obtain a longer string which matches to *something* (an identifier) if we keep reading characters.

Tokens and their inhabitation

For efficiency, most lexers usually scan their input from left to right — but only once.

Tokens and their inhabitation

For efficiency, most lexers usually scan their input from left to right — but only once.

Now — we **could** consider a language which has keywords

`For` (e.g., used in a standard 'for' loop)
`Forall` (e.g., to iterate over a data structure)

in which the language designer has decided that identifiers can't start with capital letters (such an `F`).

Tokens and their inhabitation

For efficiency, most lexers usually scan their input from left to right — but only once.

Now — we **could** consider a language which has keywords

`For` (e.g., used in a standard 'for' loop)
`Forall` (e.g., to iterate over a data structure)

in which the language designer has decided that identifiers can't start with capital letters (such an `F`). But what do we do if we encounter the code

```
Foralg = 1 to 5 do ...
```

Tokens and their inhabitation

For efficiency, most lexers usually scan their input from left to right — but only once.

Now — we **could** consider a language which has keywords

`For` (e.g., used in a standard 'for' loop)
`Forall` (e.g., to iterate over a data structure)

in which the language designer has decided that identifiers can't start with capital letters (such an `F`). But what do we do if we encounter the code

```
Foralg = 1 to 5 do ...
```

Is this a valid 'for' loop with the iterator `alg` (requiring the compiler to look at least three characters ahead to break `Foralg` into `For` and `alg`)? If not, when in reading `Foralg` does it give up on getting a keyword?

The lexer can only classify strings by **looking ahead**.

Tokens and their inhabitation

Question: how far does the lexer have to look ahead (in Java-like languages) before it can decide whether `For...` is the keyword `For`, or something else?

Tokens and their inhabitation

Question: how far does the lexer have to look ahead (in Java-like languages) before it can decide whether `For...` is the keyword `For`, or something else?

Some old languages required unbounded lookahead — but the more lookahead required, the less efficient the lexing process. Modern languages require little lookahead (typically only 1 character).

Tokens and their inhabitation

Question: how far does the lexer have to look ahead (in Java-like languages) before it can decide whether `For...` is the keyword `For`, or something else?

Some old languages required unbounded lookahead — but the more lookahead required, the less efficient the lexing process. Modern languages require little lookahead (typically only 1 character).

- "`Foralg`" example: we would give up on "`For`" after the next character '`a`', because it **might** match something else (such as `Forall`) — though that match might not happen.

In practise, to avoid problems, we define token classes so that they may be easily delimited by whitespace or punctuation (used as token boundaries).

Summary

The goal of lexical analysis is:

Summary

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g., whitespace, syntax of keywords, ...)

Summary

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g., whitespace, syntax of keywords, ...)
- Give a rough classification (simplification) of the input to simplify the next stage (parsing). In detail:

Summary

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g., whitespace, syntax of keywords, ...)
- Give a rough classification (simplification) of the input to simplify the next stage (parsing). In detail:
  - Identify the lexical structure and tokens of the language.

Summary

The goal of lexical analysis is:

- Get rid of semantically irrelevant information (e.g., whitespace, syntax of keywords, ...)
- Give a rough classification (simplification) of the input to simplify the next stage (parsing). In detail:
  - Identify the lexical structure and tokens of the language.
  - Partition the input string into small units (tokens) used by the parser.

Lexing does a left-to-right scan of the input string, sometimes with lookahead.

The material in the textbooks

- Dragon Book: Chapter 2.6 and Chapter 3.
- Appel, Palsberg: Chapter 2.1 and 2.2
- "Engineering a compiler": Chapter 2.1 and 2.2