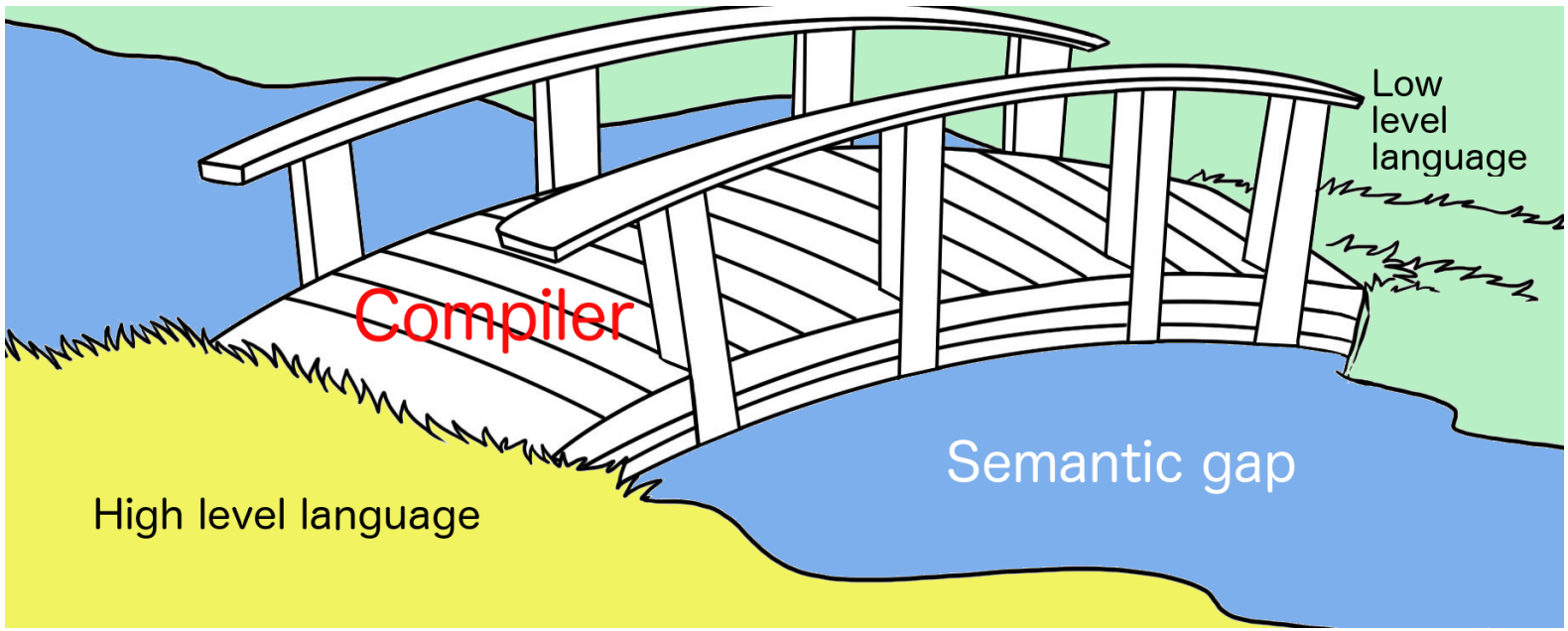
Compilers and computer architecture

8. Semantic analysis — an overview

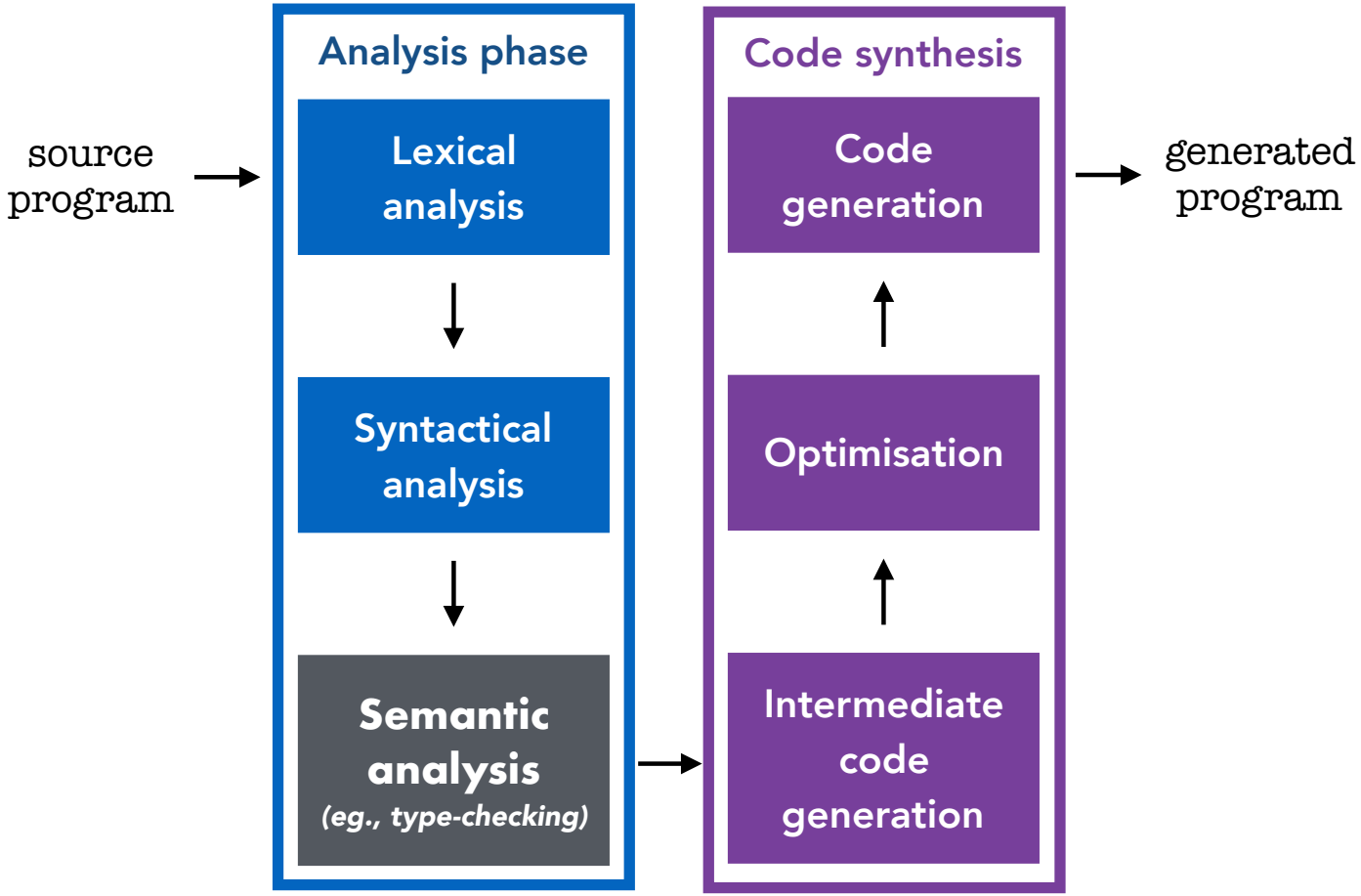
Ian Mackie

Autumn 2022

Recall the function of compilers



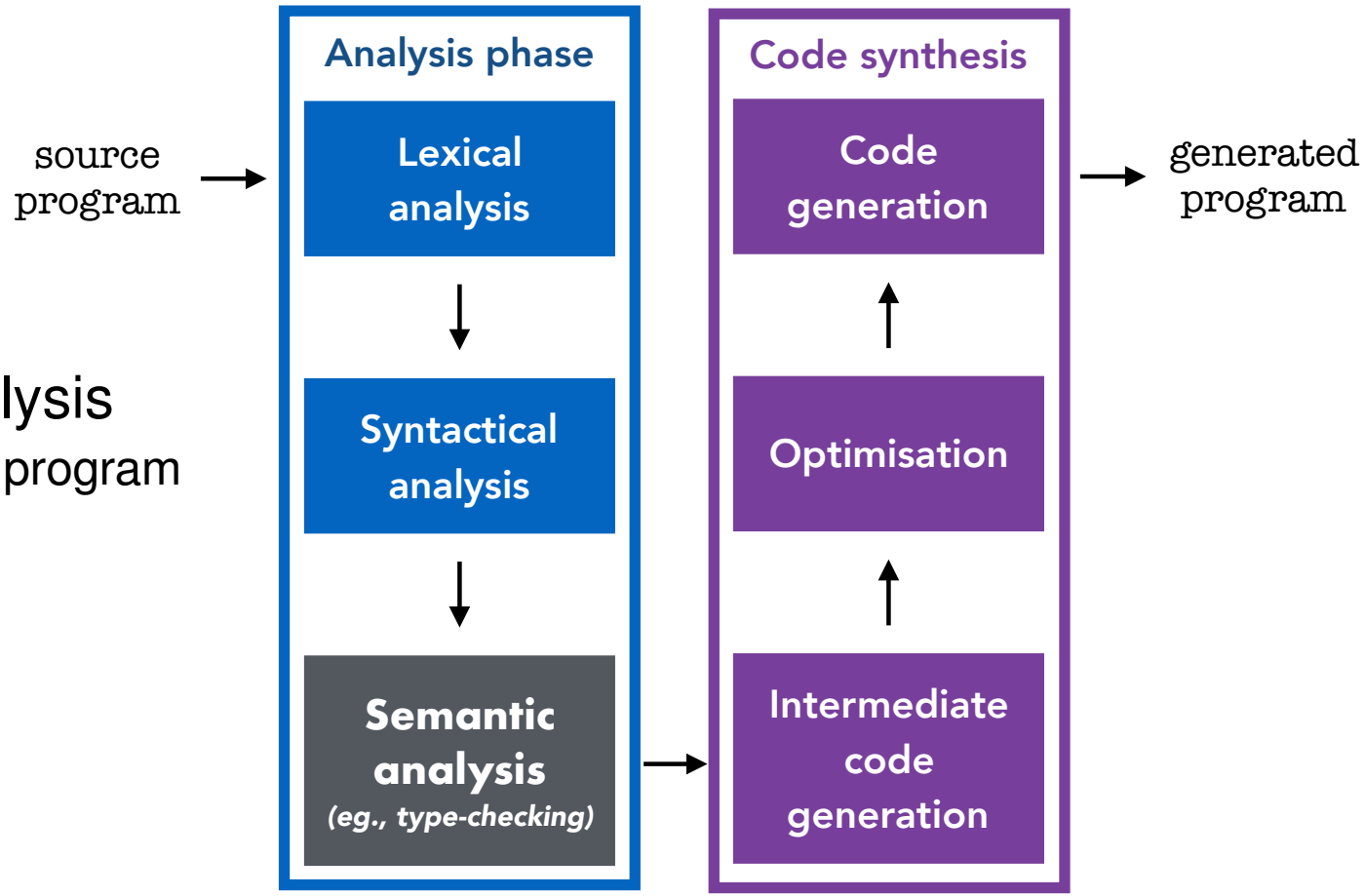
What are we up to now?



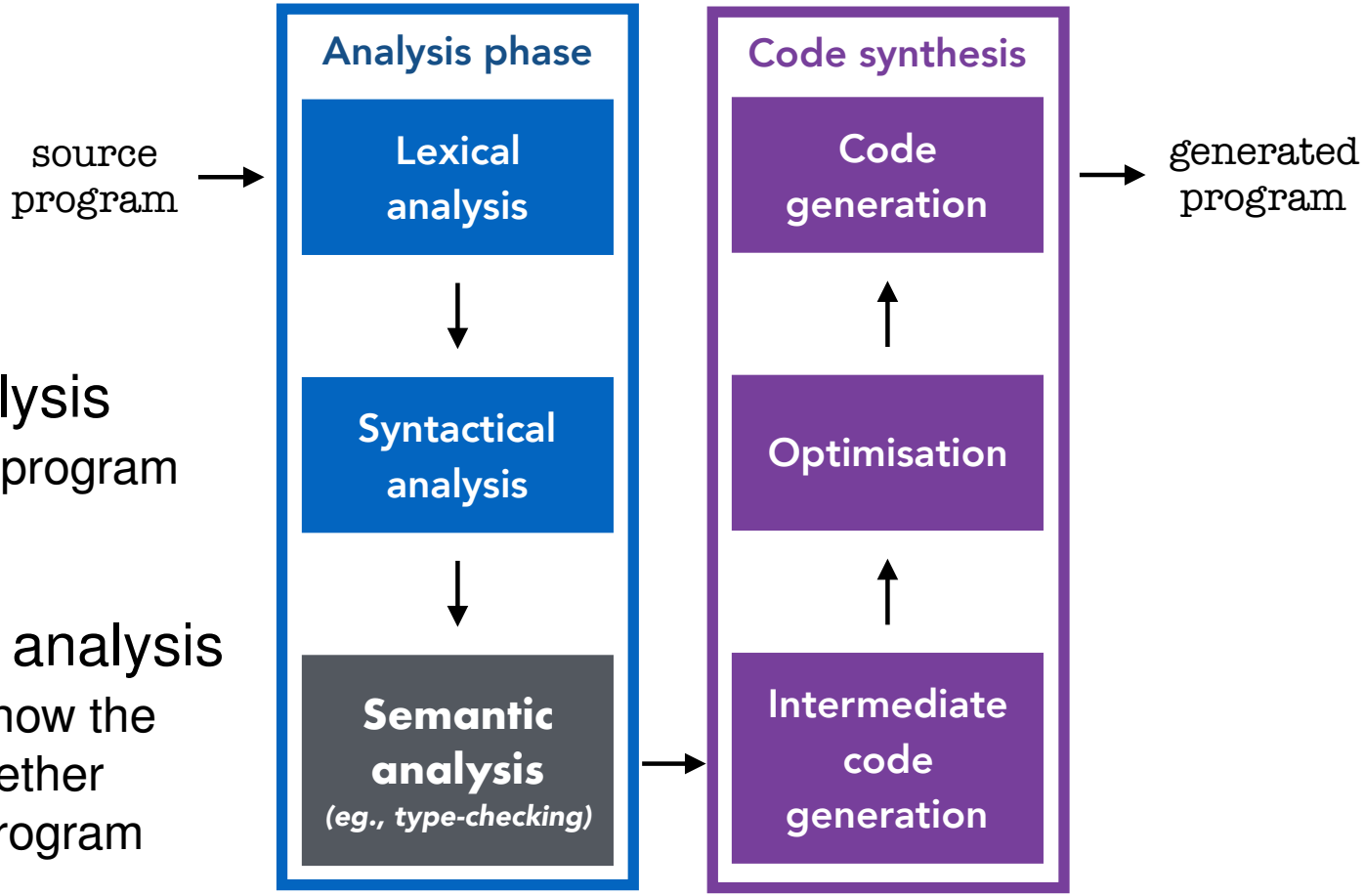
What are we up to now?

We have done

- lexical analysis
  breaking the program into tokens



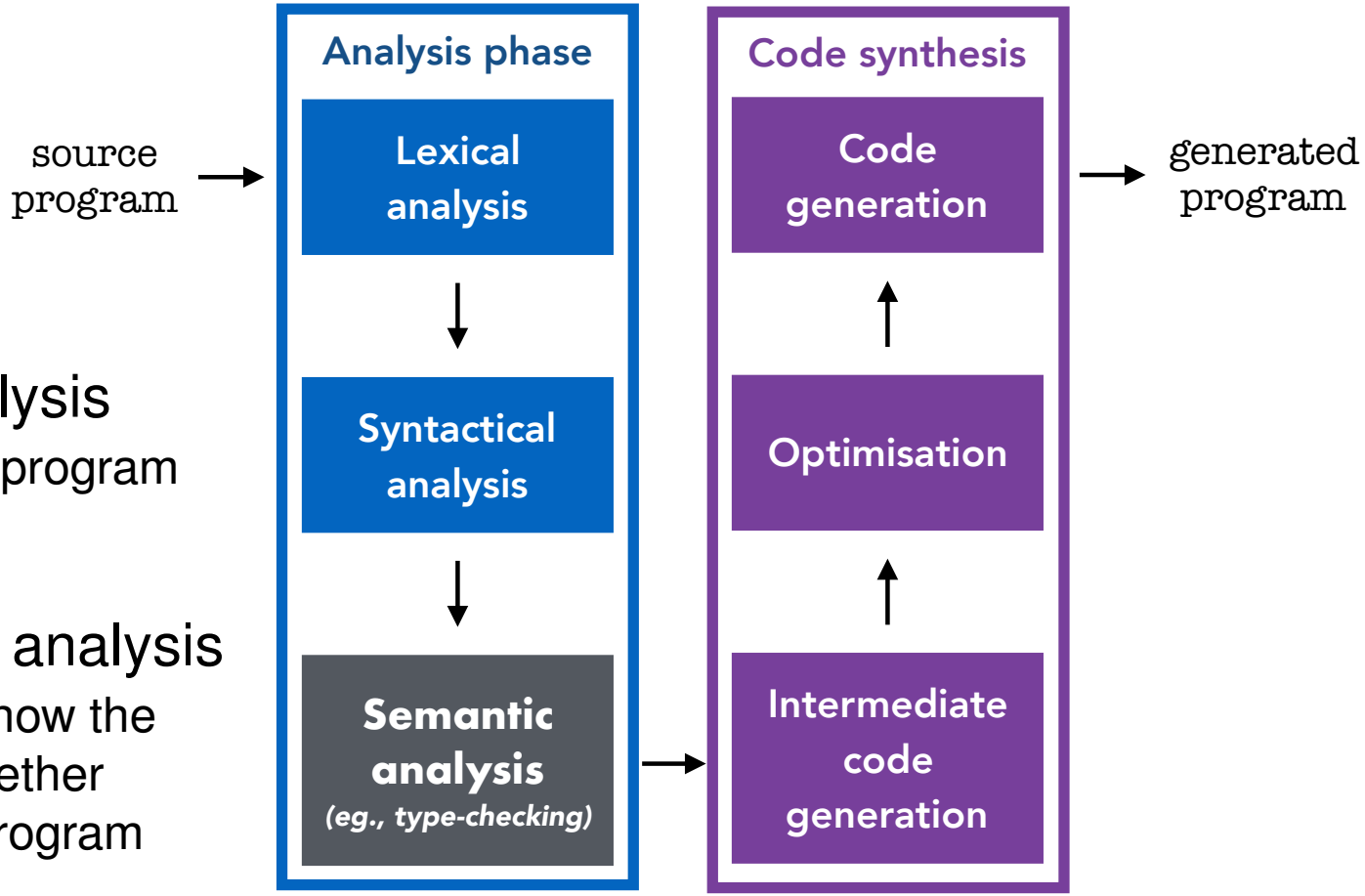
What are we up to now?



We have done

- lexical analysis
  breaking the program into tokens
- syntactical analysis
  determining how the tokens fit together to form the program structure

What are we up to now?

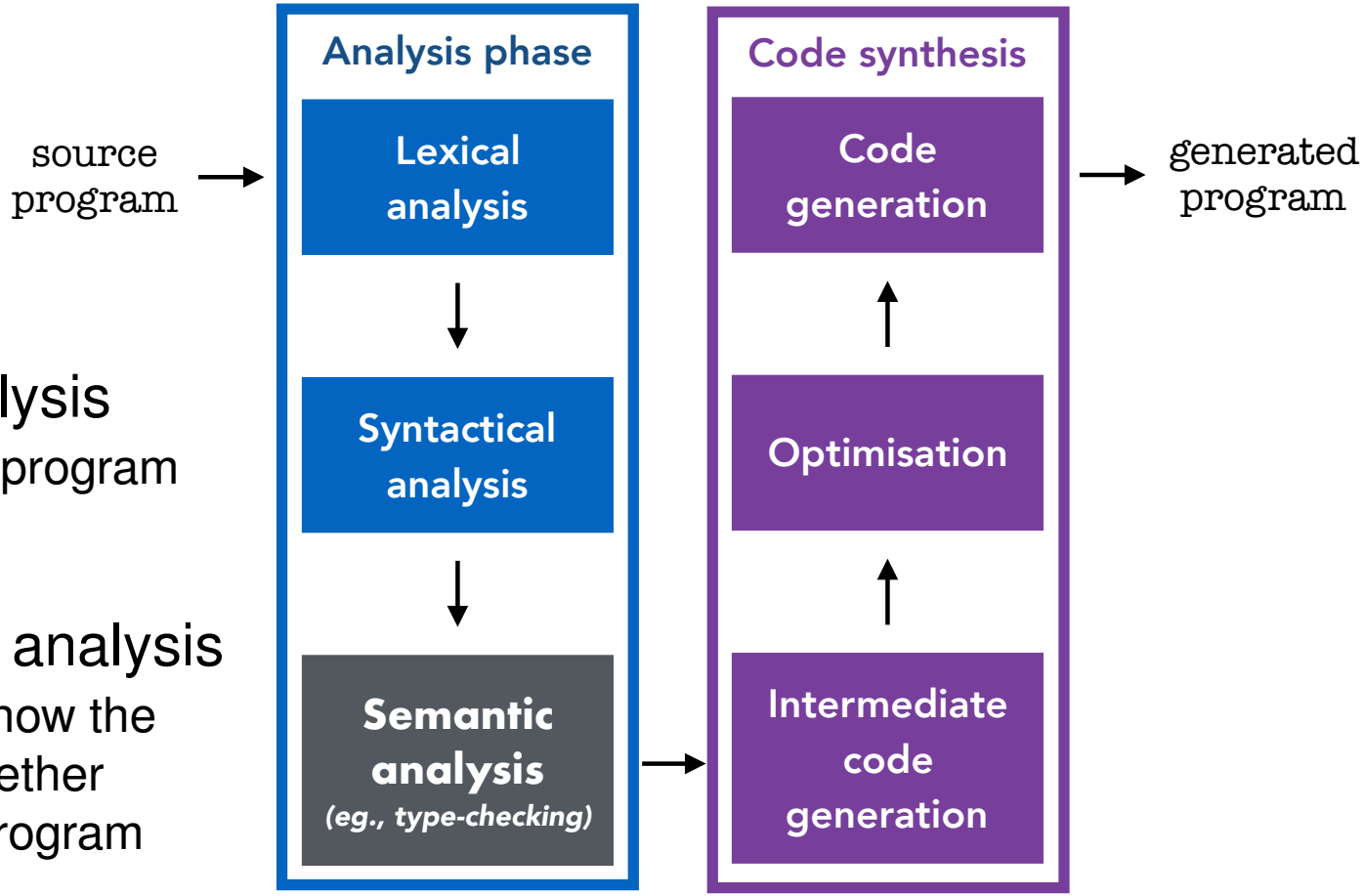


We have done

- lexical analysis
  breaking the program into tokens
- syntactical analysis
  determining how the tokens fit together to form the program structure

We are increasingly confident about the meaning of the input (*e.g.*, that it makes **any** sense).

What are we up to now?



We have done

- lexical analysis
  breaking the program into tokens
- syntactical analysis
  determining how the tokens fit together to form the program structure

We are increasingly confident about the meaning of the input (*e.g.*, that it makes **any** sense).

There is one last round of analysis: **semantic analysis**.

Semantics?

Lexical analysis: are the following valid English words?

Semantics?

Lexical analysis: are the following valid English words?

Semantics?

Lexical analysis: are the following valid English words?

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**
  — yes: all of these are *adjectives* (which modify nouns).

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**
  — yes: all of these are *adjectives* (which modify nouns).
- **eat**, **jump**, **sleep**

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**
  — yes: all of these are *adjectives* (which modify nouns).
- **eat**, **jump**, **sleep**
  — yes: these are both *verbs*.

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**
  — yes: all of these are *adjectives* (which modify nouns).
- **eat**, **jump**, **sleep**
  — yes: these are both *verbs*.
- **quickly**, **furiously**

Semantics?

Lexical analysis: are the following valid English words?

- **fox**, **box**, **frog**, **idea**, **colour**, **fury**
  — yes: all of these are *nouns*.
- **green**, **quick**, **quiet**, **colourless**, **furious**
  — yes: all of these are *adjectives* (which modify nouns).
- **eat**, **jump**, **sleep**
  — yes: these are both *verbs*.
- **quickly**, **furiously**
  — yes: these are both *adverbs* (which modify verbs).

Semantics

Semantics

Syntactic analysis: are the following sentences grammatically correct English?

Semantics

Syntactic analysis: are the following sentences grammatically correct English?

English grammar fragment:

$$\begin{aligned} \mathbf{S} &\rightarrow \mathbf{NP}\ \ \mathbf{VP} \\ \mathbf{NP} &\rightarrow \text{noun} \mid \text{adjective}\ \ \mathbf{NP} \\ \mathbf{VP} &\rightarrow \text{verb} \mid \text{verb}\ \ \mathbf{NP} \mid \text{verb}\ \ \text{adverb} \end{aligned}$$

Semantics

Syntactic analysis: are the following sentences grammatically correct English?

Semantics

Syntactic analysis: are the following sentences grammatically correct English?

- **Foxes eat boxes**

Semantics

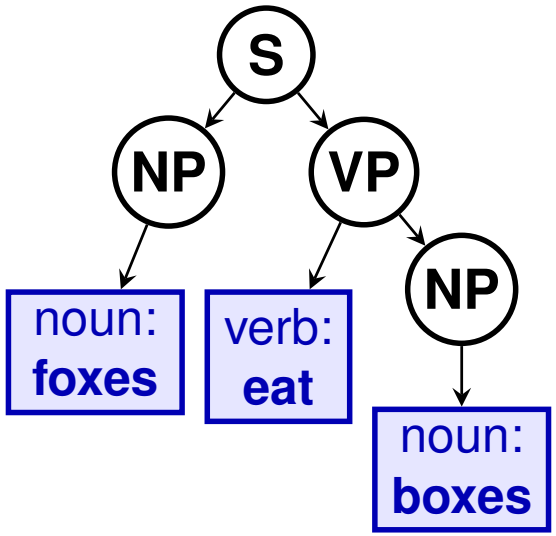
Syntactic analysis: are the following sentences grammatically correct English?

- **Foxes eat boxes**
  Yes (though it may not be true).

Semantics

Syntactic analysis: are the following sentences grammatically correct English?

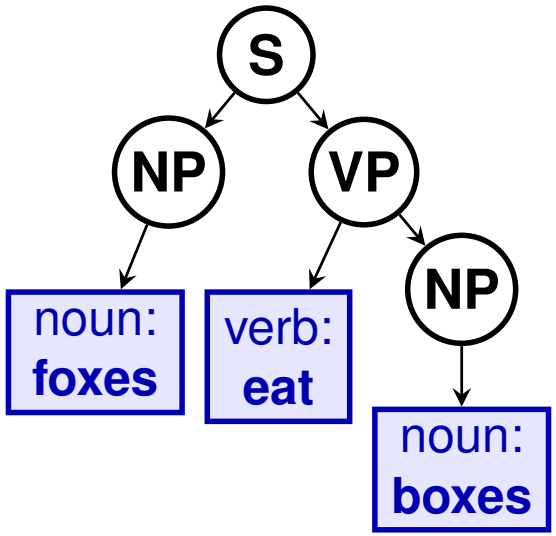
- **Foxes eat boxes**
  Yes (though it may not be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

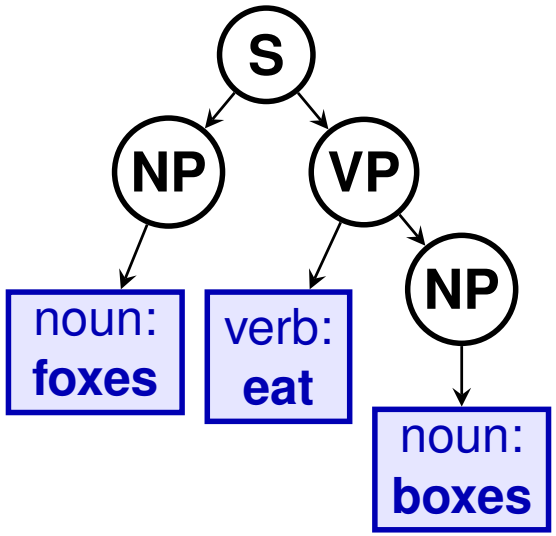
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

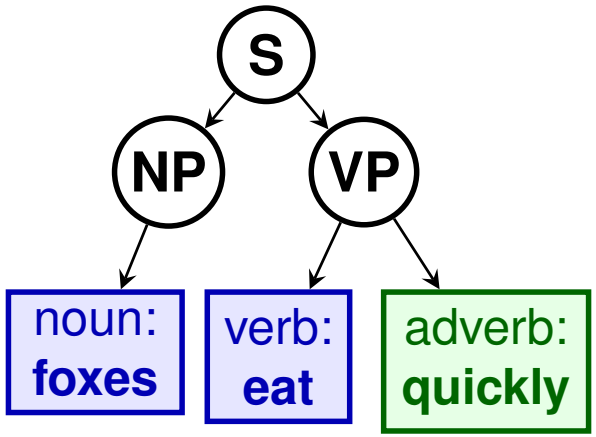
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

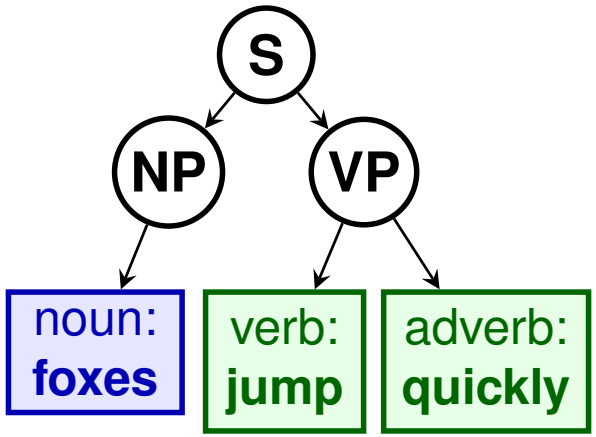
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

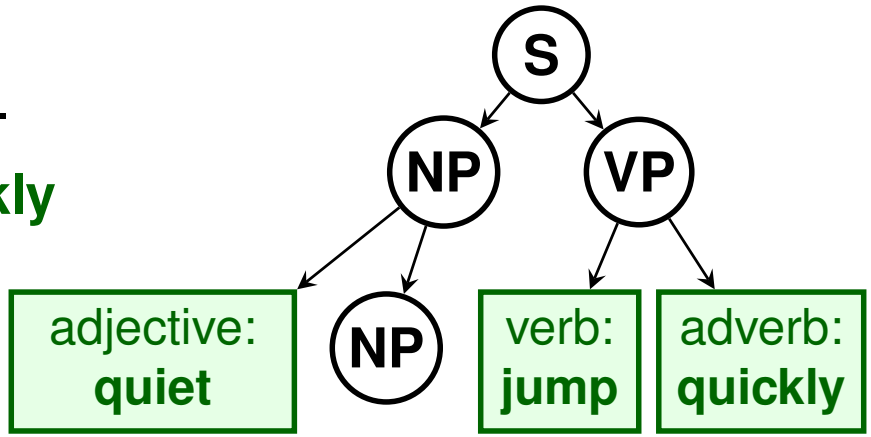
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

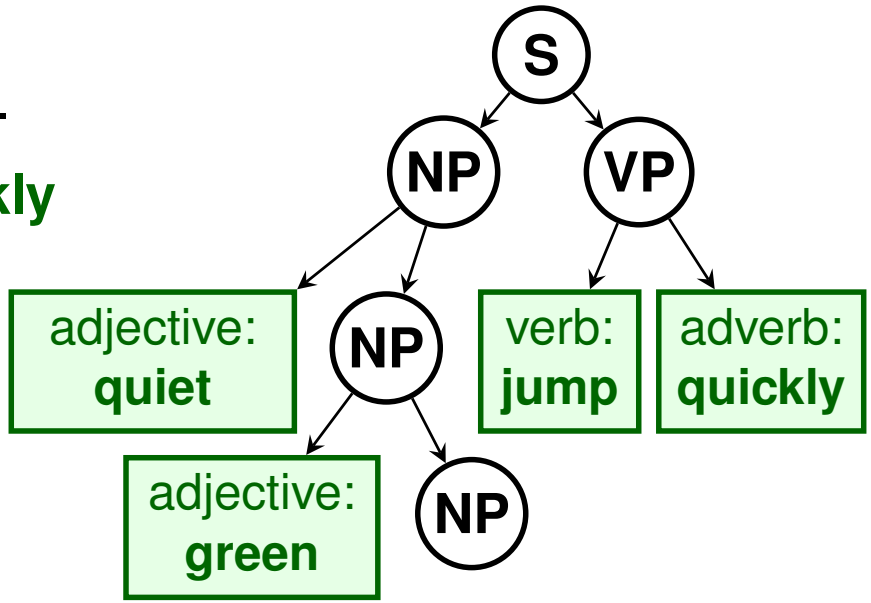
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

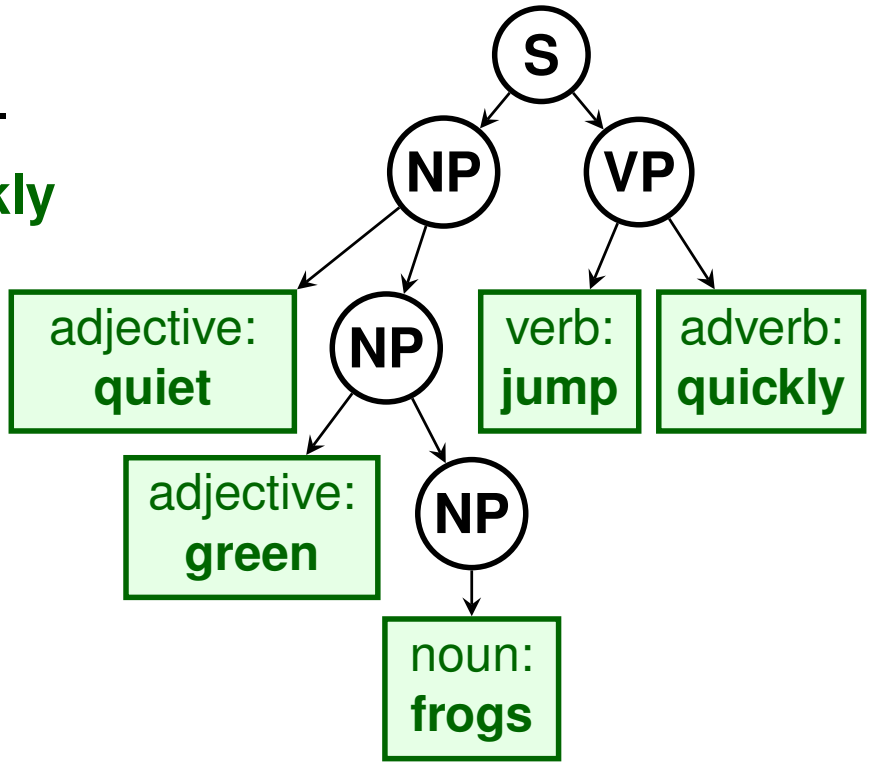
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

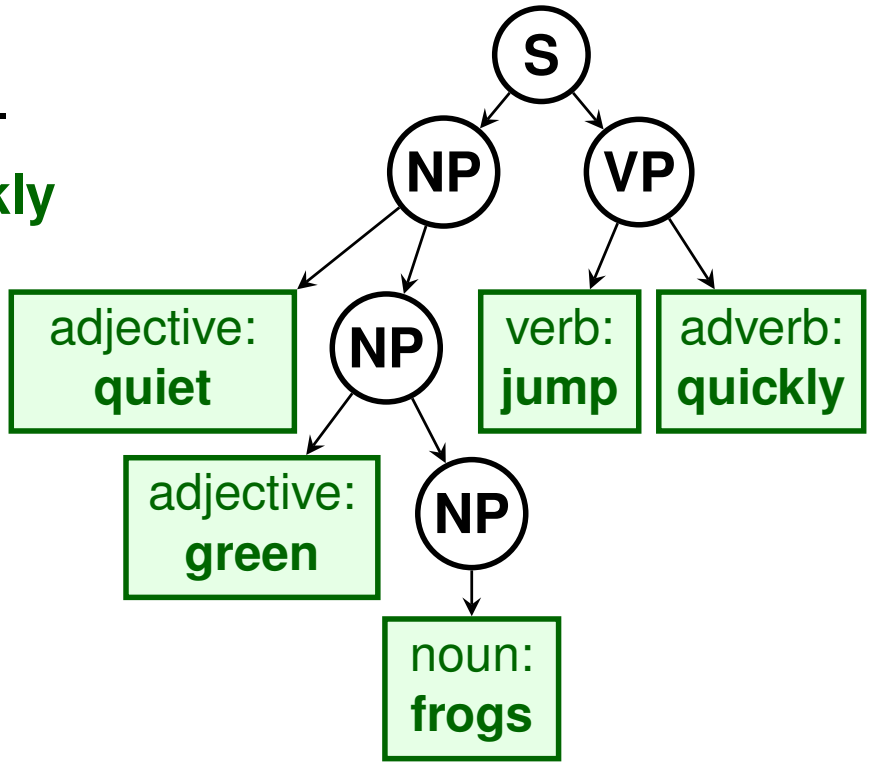
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

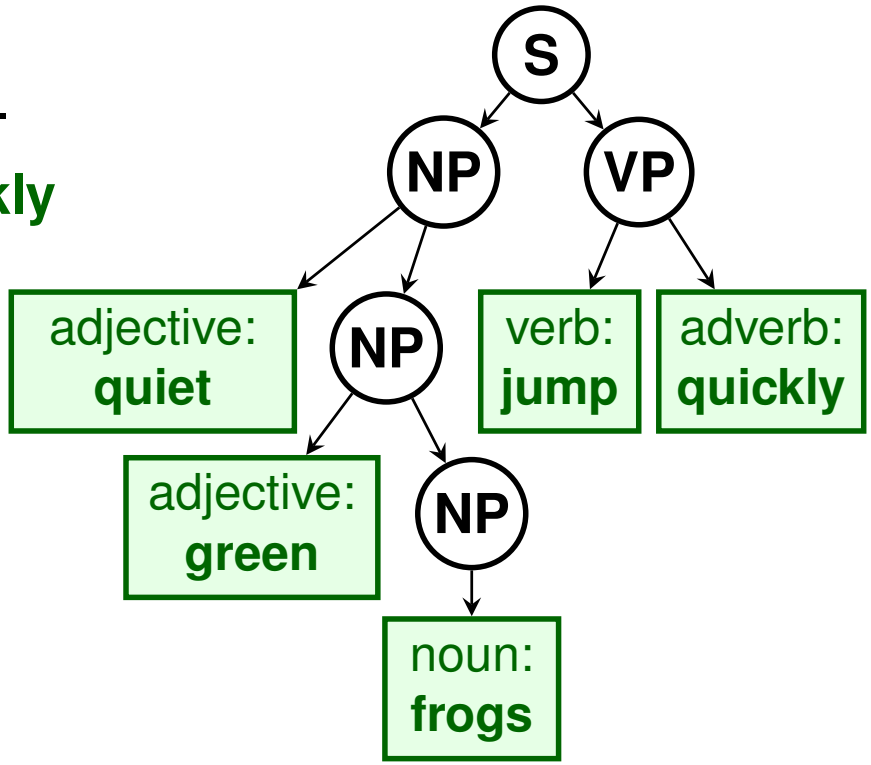
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

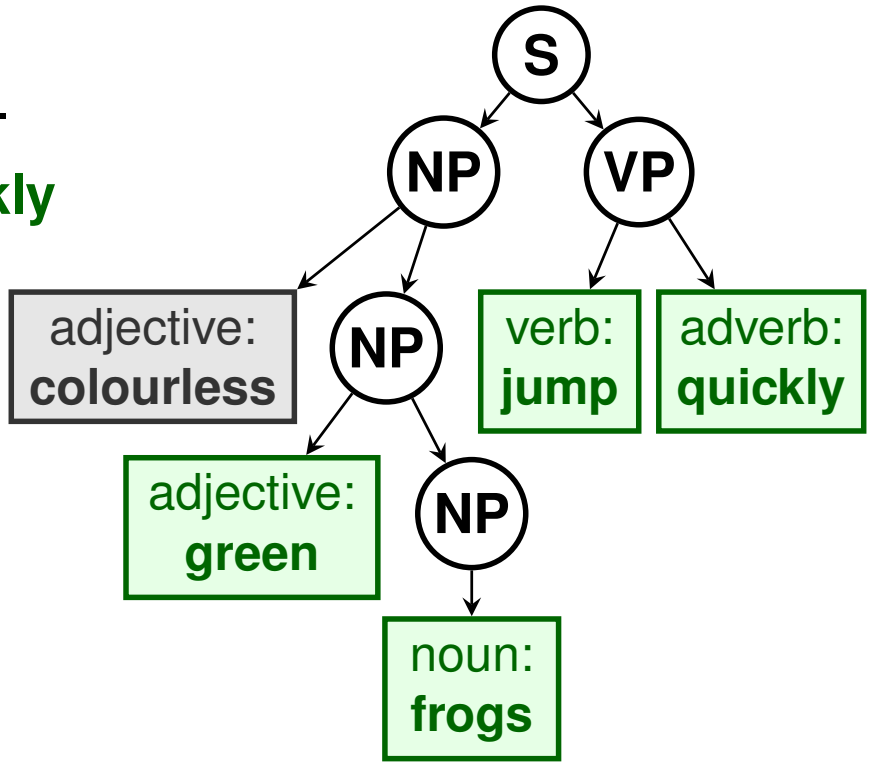
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

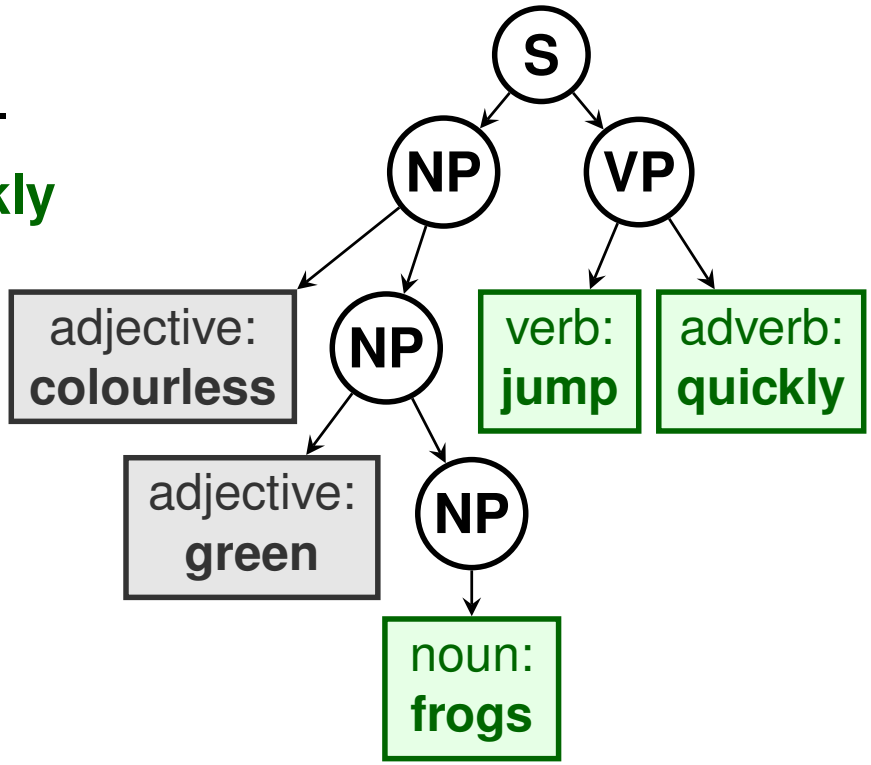
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

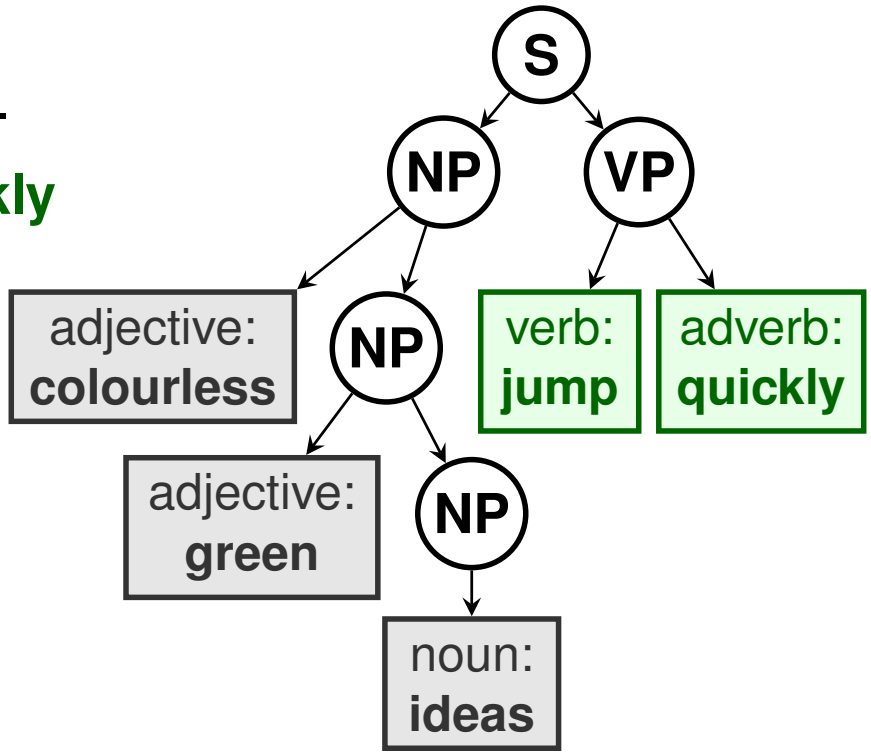
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

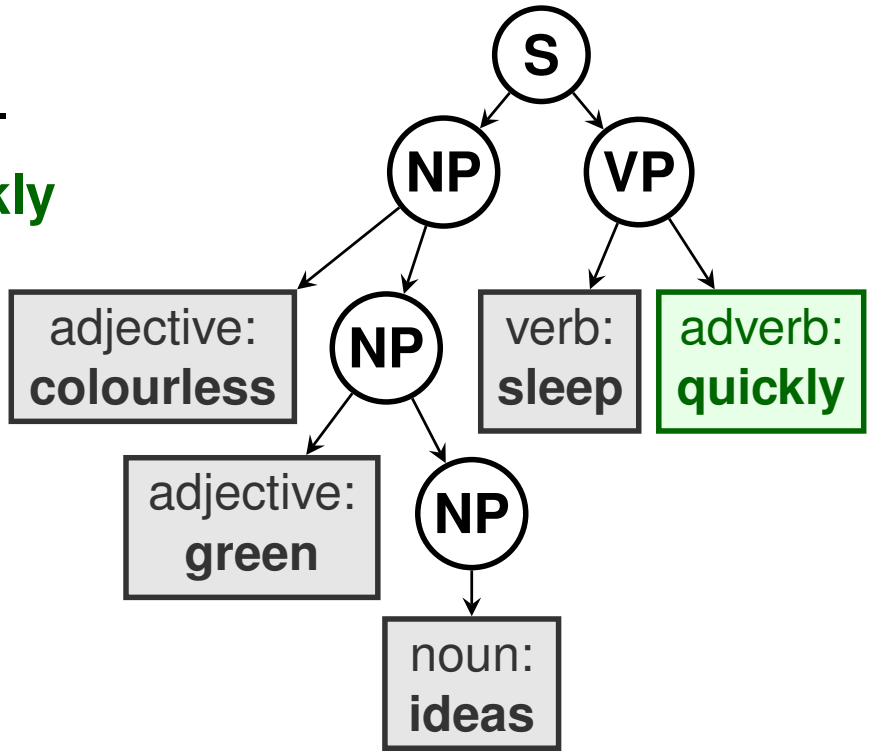
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

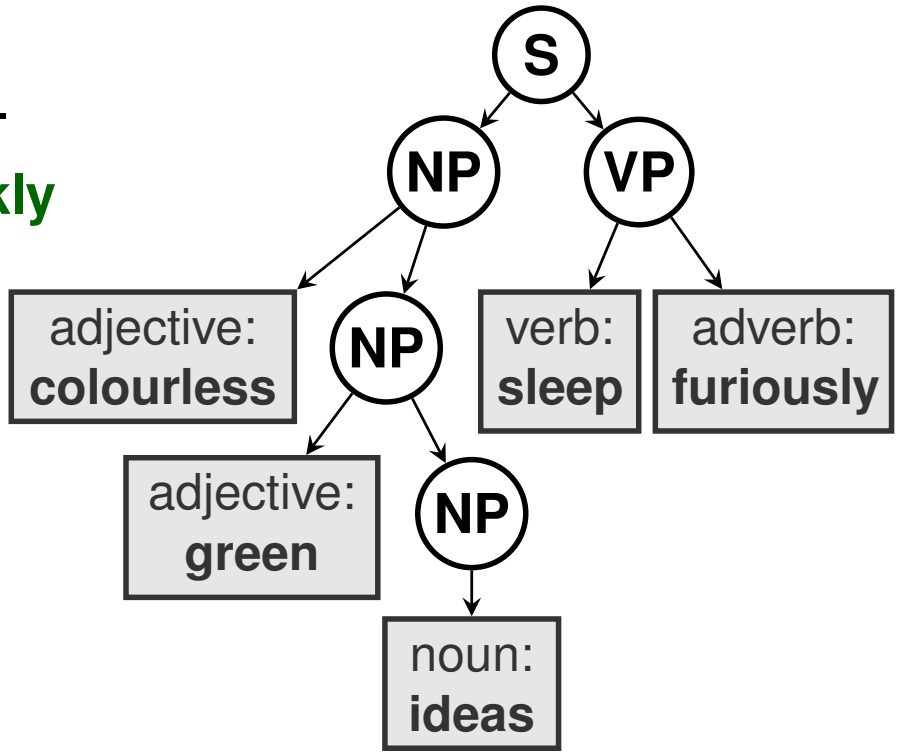
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

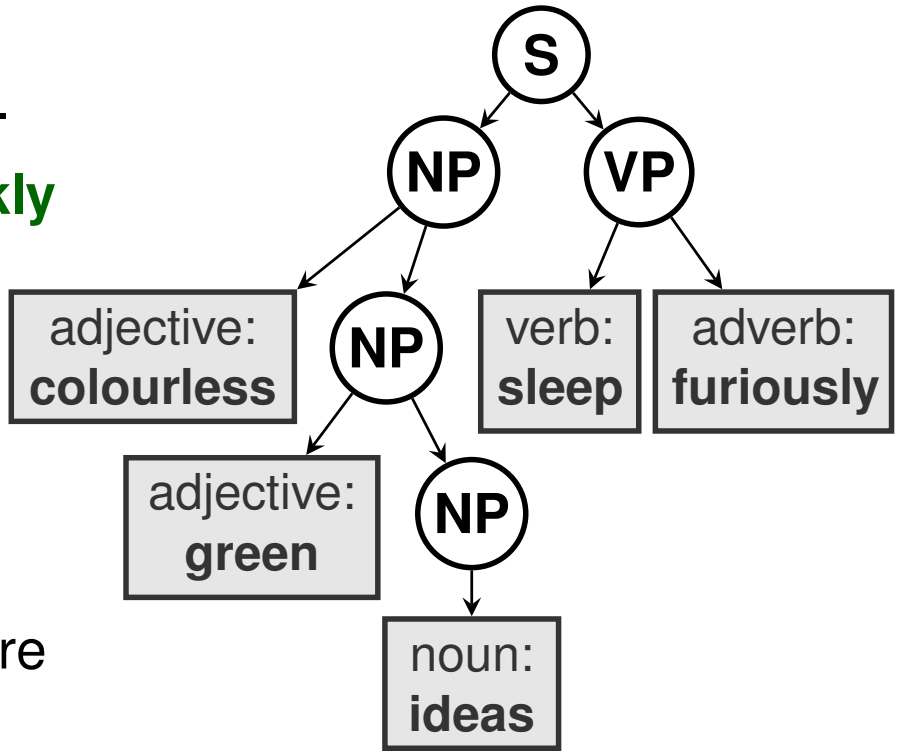
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Semantics

Syntactic analysis: are the following sentences grammatically correct English?

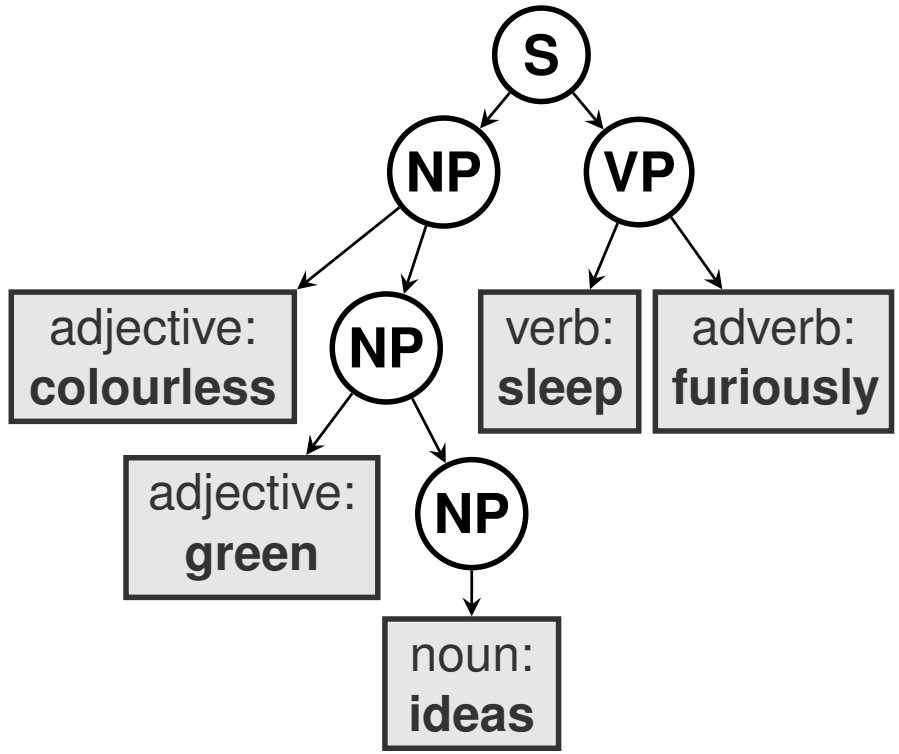
- **Foxes eat boxes**
  Yes (though it may not be true).
- **Quiet green frogs jump quickly**
  Yes (and it may even be true).
- **Colourless green ideas sleep furiously**
  Yes (incredibly!)



Grammatical correctness cannot capture truth — it cannot even ensure coherent ideas

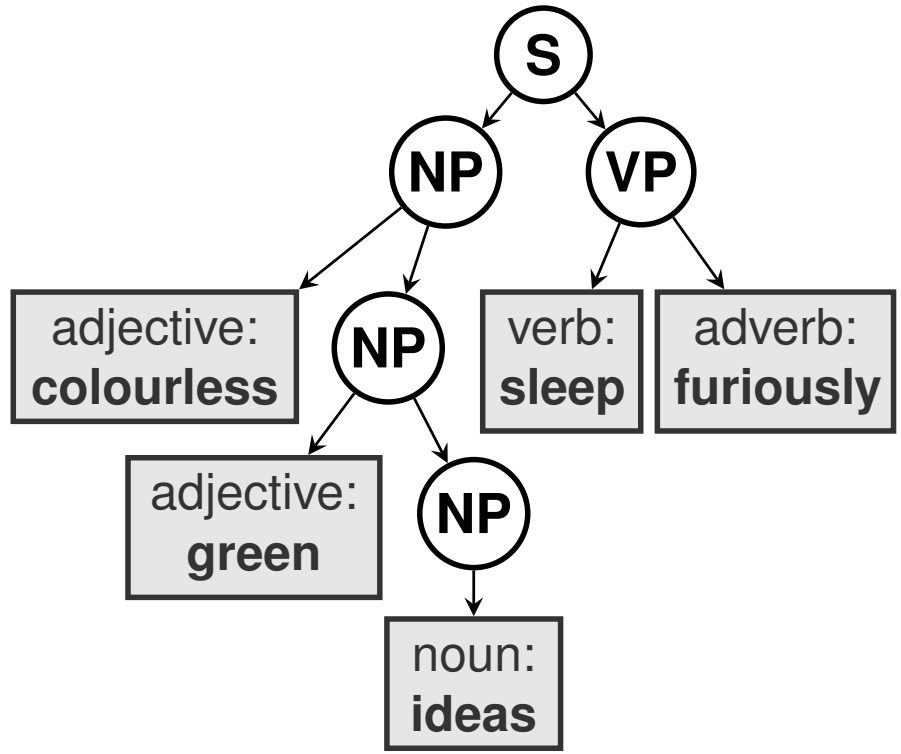
CFGs are "memoryless"

Semantics


S
NP
VP
adjective:
colourless
NP
verb:
sleep
adverb:
furiously
adjective:
green
NP
noun:
ideas

Semantics

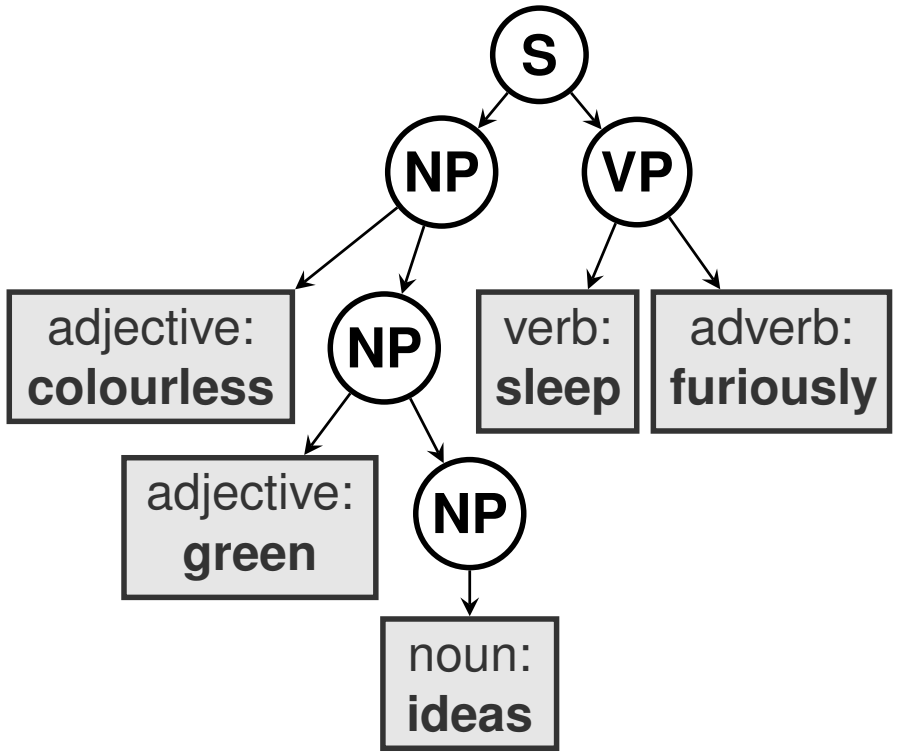
Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’



Semantics

Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’

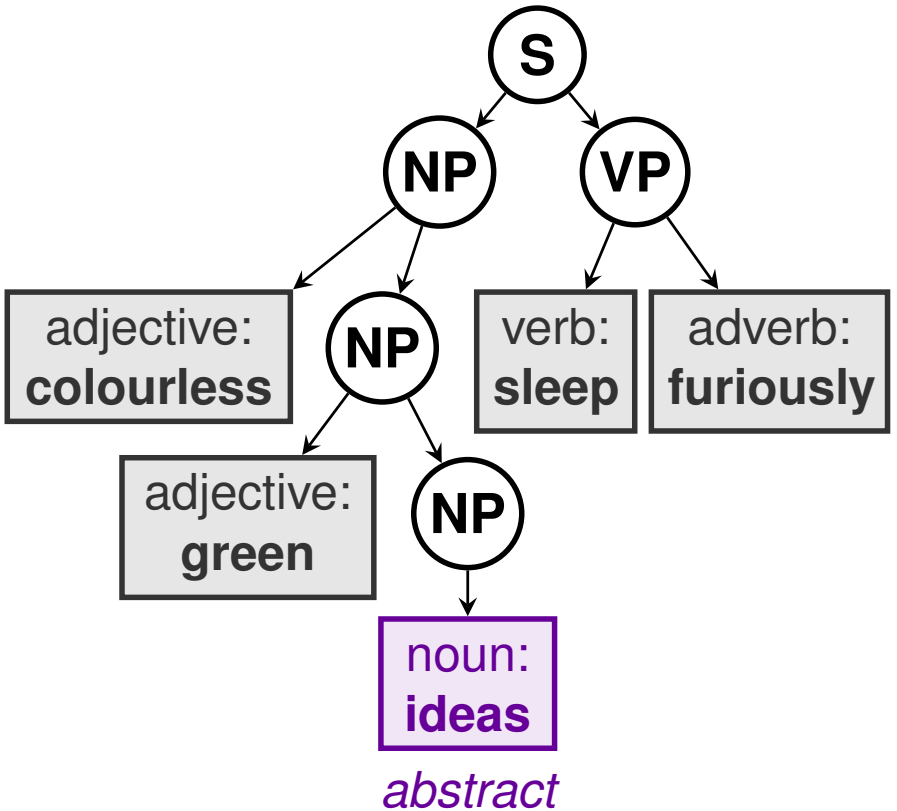
We classify the words, and traverse the syntax tree, to check for coherence.



Semantics

Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’

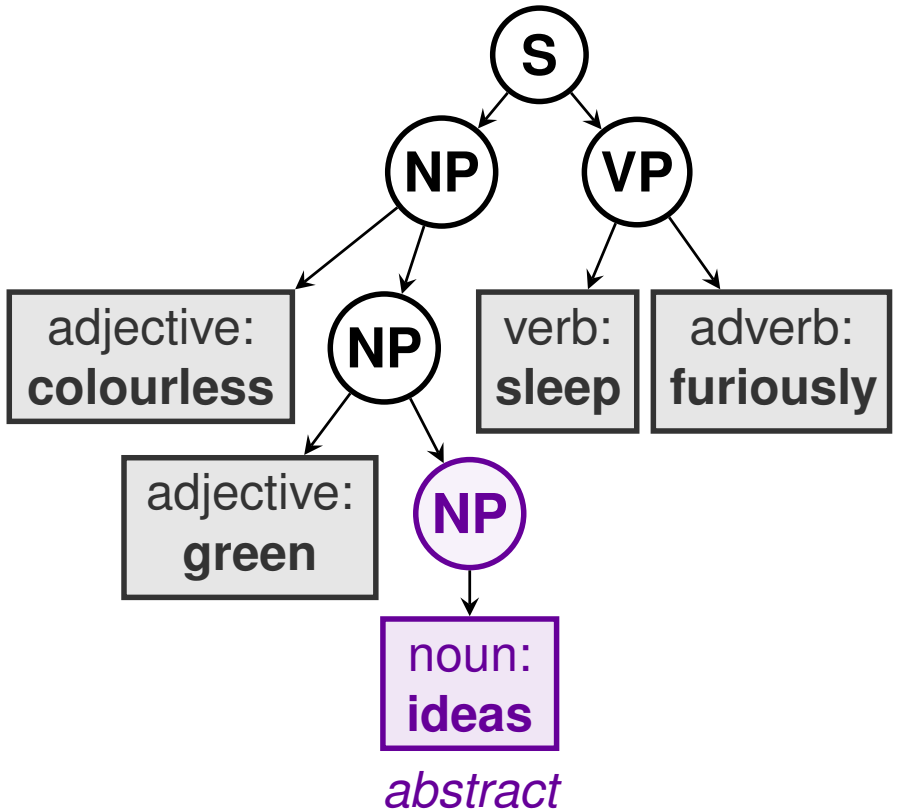
We classify the words, and traverse the syntax tree, to check for coherence.



Semantics

Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’

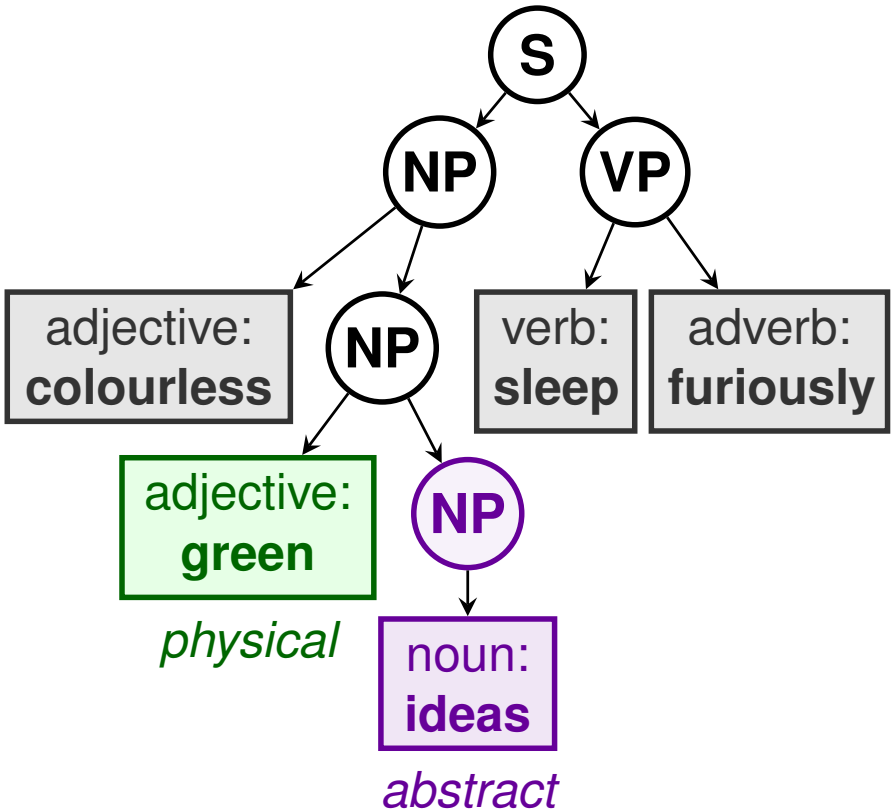
We classify the words, and traverse the syntax tree, to check for coherence.



Semantics

Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’

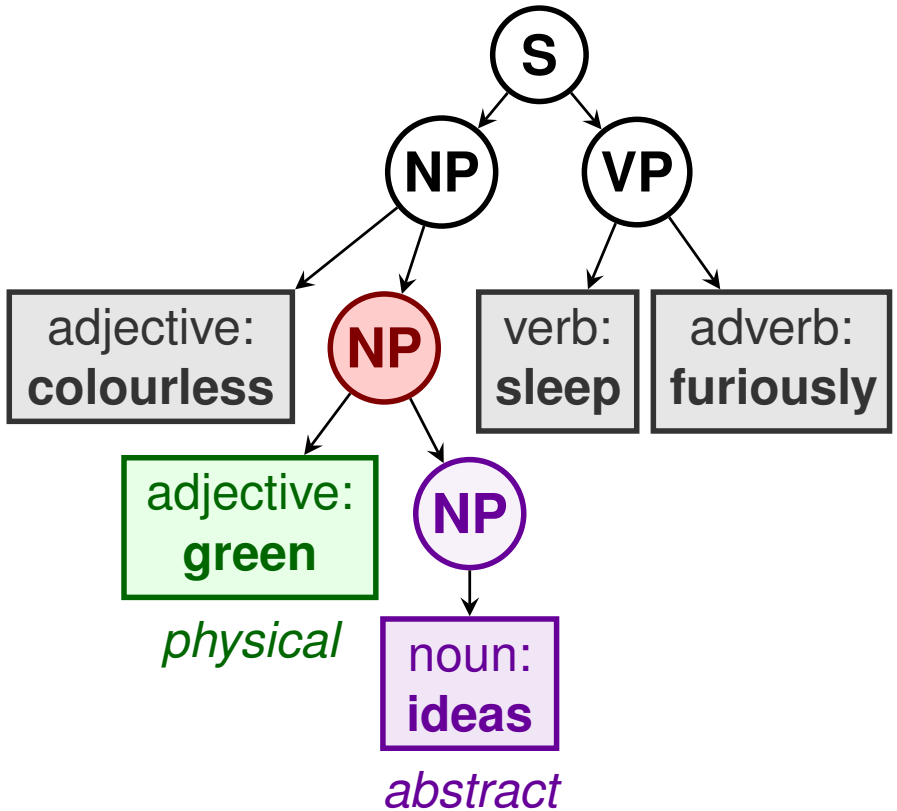
We classify the words, and traverse the syntax tree, to check for coherence.



Semantics

Semantic analysis: relies on classification of terms, to ensure some minimum standard of ‘coherence’

We classify the words, and traverse the syntax tree, to check for coherence.



What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

- Syntactic phase: local *recursive* analysis

  (uses the token sequence to determine a program structure; rejects if the tokens don't form a valid structure.)

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

- Syntactic phase: local *recursive* analysis

  (uses the token sequence to determine a program structure; rejects if the tokens don't form a valid structure.)

- **Semantic phase:** non-local analysis of *specification*, *e.g.*:

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

- Syntactic phase: local *recursive* analysis

  (uses the token sequence to determine a program structure; rejects if the tokens don't form a valid structure.)

- **Semantic phase:** non-local analysis of *specification*, *e.g.*:
  - identify dependencies of class / method declarations;

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

- Syntactic phase: local *recursive* analysis

  (uses the token sequence to determine a program structure; rejects if the tokens don't form a valid structure.)

- **Semantic phase:** non-local analysis of *specification*, *e.g.*:
  - identify dependencies of class / method declarations;
  - consistency of operations / method invocations;

What the analysis phase is for

The main jobs of the analysis phase are to:

1. characterise the input program; and
2. reject ill-formed inputs.

The different stages do this with increasing precision, each building on the work of the last:

- Lexical phase: highly localised analysis

  (discretises / classifies the input into tokens; rejects if the input doesn't consist only of valid tokens.)

- Syntactic phase: local *recursive* analysis

  (uses the token sequence to determine a program structure; rejects if the tokens don't form a valid structure.)

- **Semantic phase:** non-local analysis of *specification*, *e.g.*:
  - identify dependencies of class / method declarations;
  - consistency of operations / method invocations;

  — rejecting if the program lacks a sound realisation.

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**.

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**. In particular:

- it needs enough information to determine what instructions to generate;
- that information must be **consistent** throughout the program.

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**. In particular:

- it needs enough information to determine what instructions to generate;
- that information must be **consistent** throughout the program.

How the compiler obtains that information, and what standard of consistency it imposes, depends on the language.

Different standards of consistency will usually make it easier for the programmer

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**. In particular:

- it needs enough information to determine what instructions to generate;
- that information must be **consistent** throughout the program.

How the compiler obtains that information, and what standard of consistency it imposes, depends on the language.

Different standards of consistency will usually make it easier for the programmer, either

- to quickly write a program;

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**. In particular:

- it needs enough information to determine what instructions to generate;
- that information must be **consistent** throughout the program.

How the compiler obtains that information, and what standard of consistency it imposes, depends on the language.

Different standards of consistency will usually make it easier for the programmer, either

- to quickly write a program; or
- to predict the behaviour of the program.

A “sound” realisation?

The job of a compiler is to determine how a program may be **realised**. In particular:

- it needs enough information to determine what instructions to generate;
- that information must be **consistent** throughout the program.

How the compiler obtains that information, and what standard of consistency it imposes, depends on the language.

Different standards of consistency will usually make it easier for the programmer, either

- to quickly write a program; or
- to predict the behaviour of the program.

More sophisticated methods may make both easier, by putting the burden on the compiler to determine how to realise it.

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs — *e.g.*, whether the program:

- ever halts / loops forever

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs — *e.g.*, whether the program:

- ever halts / loops forever
- uses more than 64 MB of memory

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs — *e.g.*, whether the program:

- ever halts / loops forever
- uses more than 64 MB of memory
- prints `"hello world"`

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs — *e.g.*, whether the program:

- ever halts / loops forever
- uses more than 64 MB of memory
- prints `"hello world"`
- ever stores anything other than an integer in a given variable

Rice's Theorem and undecidability

There are limits to what we can achieve in the name of predictability:

Rice's theorem. No interesting property of programs (*i.e.*, program execution) is decidable for all computer programs — *e.g.*, whether the program:

- ever halts / loops forever
- uses more than 64 MB of memory
- prints `"hello world"`
- ever stores anything other than an integer in a given variable

We can answer these questions for *some* programs — usually because we write programs in such a way that we can tell the answer.

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

One common solution:

- Require that all variables be typed

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

One common solution:

- Require that all variables be typed
- Require that all assignments be well-typed

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

One common solution:

- Require that all variables be typed
- Require that all assignments be well-typed
- Check whether `x` is an integer

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

One common solution:

- Require that all variables be typed
- Require that all assignments be well-typed
- Check whether `x` is an integer
- Check whether `x` is ever initialised

Rice's Theorem and undecidability

Most programming languages impose constraints on programs, to make some of these questions easier to answer

But how do you make an undecidable problem "easier"?

Answer. By not trying very hard, for any program for which the problem is too difficult to answer.

Example. "Is the value stored in a variable `x` an integer?"

One common solution:

- Require that all variables be typed
- Require that all assignments be well-typed
- Check whether `x` is an integer
- Check whether `x` is ever initialised

— takes the shortcut of rejecting any input for which the question could possibly be difficult.

What do we want to predict?

— well, everything.

What do we want to predict?

— well, everything.

But we'll settle for predicting features of programs,
that allow us to build other programs.

What do we want to predict?

— well, everything.

But we'll settle for predicting features of programs, that allow us to build other programs.

For instance:

- Typing —
  - Data stored by an object / datatype
  - Data returned by a method / procedure
  - Ability to compose / perform a statement

What do we want to predict?

— well, everything.

But we'll settle for predicting features of programs,
that allow us to build other programs.

For instance:

- Typing —
  - Data stored by an object / datatype
  - Data returned by a method / procedure
  - Ability to compose / perform a statement
- Scope / encapsulation / modularisation —
  - Information which *isn't* affected by programs
  - Limitations to ways that data can be transformed
  - Exit points of a procedure (including exception handling)

What do we want to predict?

— well, everything.

But we'll settle for predicting features of programs,
that allow us to build other programs.

For instance:

- Typing —
  - Data stored by an object / datatype
  - Data returned by a method / procedure
  - Ability to compose / perform a statement
- Scope / encapsulation / modularisation —
  - Information which *isn't* affected by programs
  - Limitations to ways that data can be transformed
  - Exit points of a procedure (including exception handling)

A good programming language presents a compromise
between **flexibility** and **predictability** for the programmer.
Semantic analysis enforces the promised "predictability".

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?
- Are classes and variables defined only once?

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?
- Are classes and variables defined only once?
- Do the inheritance relationships make sense?

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?
- Are classes and variables defined only once?
- Do the inheritance relationships make sense?
- Methods defined only once?

Examples of “semantic analysis”

The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?
- Are classes and variables defined only once?
- Do the inheritance relationships make sense?
- Methods defined only once?
- Are private methods / members only used within the defining class?

Examples of “semantic analysis”

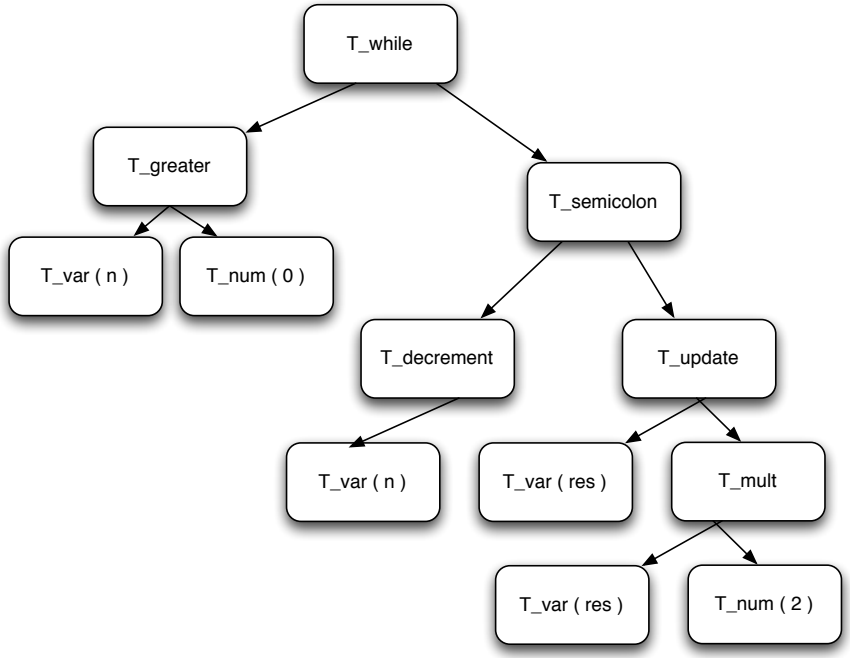
The precise requirements depend on the language —
and the mechanisms which the language uses,
to allow programs to be predictable to some extent.

Examples.

- All identifiers declared and initialised before use?
- Are all types correctly declared?
- Are classes and variables defined only once?
- Do the inheritance relationships make sense?
- Methods defined only once?
- Are private methods / members only used within the defining class?
- Are all expressions correctly typed?

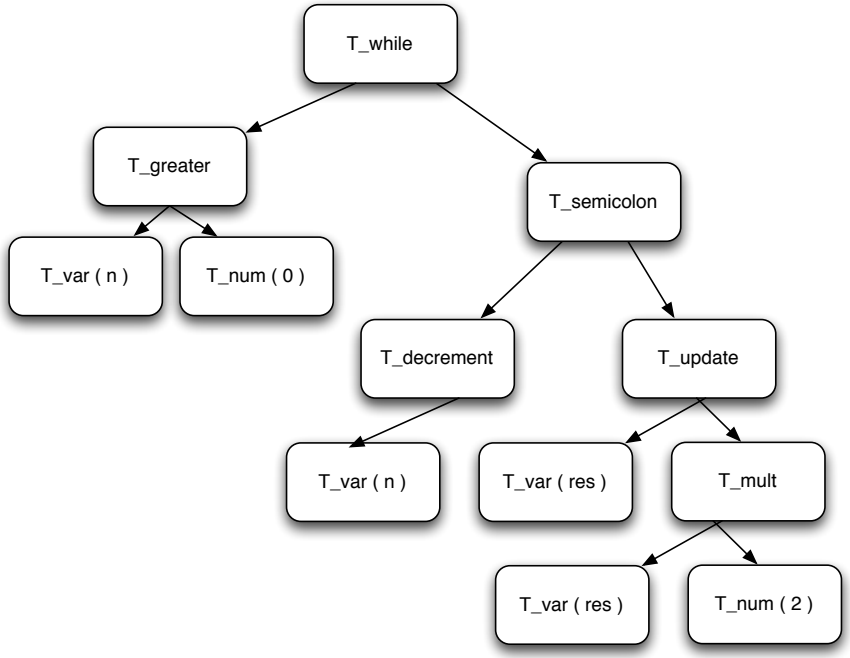
Approach to semantic analysis

As with our sketch involving English sentences, we perform semantic analysis by traversing the (abstract) syntax tree.



Approach to semantic analysis

As with our sketch involving English sentences, we perform semantic analysis by traversing the (abstract) syntax tree.



We perform the following for each node *N*, recursively:

- Perform semantic analysis on the children of *N*.
- Using the information obtained from the children, analyse the node *N*
  (*e.g.*, for type-checking, to check for unresolved identifiers, *etc.*)

Semantic analysis by traversal of the AST

The way we traverse the AST will be different, for different languages.

Semantic analysis by traversal of the AST

The way we traverse the AST will be different, for different languages.

- In many programming languages, type-checking / checking that identifiers are declared before used, depends only on what came before — AST traversal has an **in-order** quality.

Semantic analysis by traversal of the AST

The way we traverse the AST will be different, for different languages.

- In many programming languages, type-checking / checking that identifiers are declared before used, depends only on what came before — AST traversal has an **in-order** quality.
- In Java and some other languages, classes / methods may be used before they are declared. In a language such as this, a single in-order traversal is not enough

Semantic analysis by traversal of the AST

The way we traverse the AST will be different, for different languages.

- In many programming languages, type-checking / checking that identifiers are declared before used, depends only on what came before — AST traversal has an **in-order** quality.
- In Java and some other languages, classes / methods may be used before they are declared. In a language such as this, a single in-order traversal is not enough — we must:
  - Walk the AST once collecting class definitions, then **a second time** to check if each use of a class (type) identifier has a definition somewhere.

Semantic analysis by traversal of the AST

The way we traverse the AST will be different, for different languages.

- In many programming languages, type-checking / checking that identifiers are declared before used, depends only on what came before — AST traversal has an **in-order** quality.
- In Java and some other languages, classes / methods may be used before they are declared. In a language such as this, a single in-order traversal is not enough — we must:
  - Walk the AST once collecting class definitions, then **a second time** to check if each use of a class (type) identifier has a definition somewhere.
  - Alternatively, propagate information about needed definitions up in a clever way, and check if everything is OK.

Key tool: types

Types and typing systems are the key tool for semantic analysis. — But what are types?

Key tool: types

Types and typing systems are the key tool for semantic analysis. — But what are types?

Types are a **rough classification** of programs that rule out certain errors, *e.g.*, `3.4 / "hello"`.

Key tool: types

Types and typing systems are the key tool for semantic analysis. — But what are types?

Types are a **rough classification** of programs that rule out certain errors, *e.g.*, `3.4 / "hello"`.

Each type `T` is associated to a set of values, and operators that we can apply to values of type `T`. Conversely, with each operator with associate types that describe the nature of the operator's arguments and result.

Types as two-version programming

In mainstream PLs, types are weak (*i.e.*, vague or abstract) specifications of programs — **partial** specifications.

Types as two-version programming

In mainstream PLs, types are weak (*i.e.*, vague or abstract) specifications of programs — **partial** specifications.

In languages such as Java, programs are **annotated** with types. This can be seen as a weak form of two-version programming

Types as two-version programming

In mainstream PLs, types are weak (*i.e.*, vague or abstract) specifications of programs — **partial** specifications.

In languages such as Java, programs are **annotated** with types. This can be seen as a weak form of two-version programming — the programmer specifies what the program should do in two different ways:

- once by the actual code,
- and a second time through the types.

This redundancy is a *feature*.

Types as two-version programming

In mainstream PLs, types are weak (*i.e.*, vague or abstract) specifications of programs — **partial** specifications.

In languages such as Java, programs are **annotated** with types. This can be seen as a weak form of two-version programming — the programmer specifies what the program should do in two different ways:

- once by the actual code,
- and a second time through the types.

This redundancy is a *feature*. By providing two specifications, the probability that we make the same mistake in both expressions is lower than if we state our intention only once.

Types as two-version programming

In mainstream PLs, types are weak (*i.e.*, vague or abstract) specifications of programs — **partial** specifications.

In languages such as Java, programs are **annotated** with types. This can be seen as a weak form of two-version programming — the programmer specifies what the program should do in two different ways:

- once by the actual code,
- and a second time through the types.

This redundancy is a *feature*. By providing two specifications, the probability that we make the same mistake in both expressions is lower than if we state our intention only once.

The key idea behind semantic analysis is to look for **contradictions** between the two specifications, and reject programs with such contradictions.

Limitations of types

Type systems can protect against some inconsistencies in the program specification — but not all. The more rigorous a type system a programming language has, the harder it is to use.

Limitations of types

Type systems can protect against some inconsistencies in the program specification — but not all. The more rigorous a type system a programming language has, the harder it is to use.

One example of a specification which most type systems generally do not enforce, is bounds checking on arrays:

```
int [] a = new int [ 10 ];
a [ 20 ] = 3;
```

Limitations of types

Type systems can protect against some inconsistencies in the program specification — but not all. The more rigorous a type system a programming language has, the harder it is to use.

One example of a specification which most type systems generally do not enforce, is bounds checking on arrays:

```
int [] a = new int [ 10 ];
a [ 20 ] = 3;
```

- This is caught by some languages at run-time, and by some languages it is not caught at all.

Limitations of types

Type systems can protect against some inconsistencies in the program specification — but not all. The more rigorous a type system a programming language has, the harder it is to use.

One example of a specification which most type systems generally do not enforce, is bounds checking on arrays:

```
int [] a = new int [ 10 ];
a [ 20 ] = 3;
```

- This is caught by some languages at run-time, and by some languages it is not caught at all.
- For a type system to catch this:
  - How would having array length be part of the type system, affect the way that methods take arrays as arguments?
  - How do we check if an array indexed by an *iterator* is "well-typed"?

Static typing and dynamic typing

An important distinction is that between:

- **statically typed languages** — *e.g.*, Java, C, C++
- **dynamically typed languages** — *e.g.*, Python, Javascript

Static typing and dynamic typing

An important distinction is that between:

- **statically typed languages** — *e.g.*, Java, C, C++
  - type-checking happens at compile-time
- **dynamically typed languages** — *e.g.*, Python, Javascript

Static typing and dynamic typing

An important distinction is that between:

- **statically typed languages** — *e.g.*, Java, C, C++
  - type-checking happens at compile-time
- **dynamically typed languages** — *e.g.*, Python, Javascript
  - type-checking happens at run-time — at the last possible moment. For example: just before we execute `x+2` , we check that `x` contains an integer.

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantages for dynamically typed languages:
  - More flexibility. There are many programs that are safe, but cannot be typed at compile time,

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantages for dynamically typed languages:
  - More flexibility. There are many programs that are safe, but cannot be typed at compile time, *e.g.*,

```
x = 100
x = x*x
println ( x+1 )
```

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantages for dynamically typed languages:
  - More flexibility. There are many programs that are safe, but cannot be typed at compile time, *e.g.*,

```
x = 100
x = x*x
println ( x+1 )
x = "hello"
x = concatenate( x, " world" )
println ( x )
```

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantages for dynamically typed languages:
  - More flexibility. There are many programs that are safe, but cannot be typed at compile time, *e.g.*,

```
x = 100
x = x*x
println ( x+1 )
x = "hello"
x = concatenate( x, " world" )
println ( x )
```

  - Moreover, vis-a-vis languages like Java, we don't have to write type annotations. Less work . . .

Dynamic vs static typing

- Disadvantages for dynamically typed languages:
  - Slow, because of constant type-checking at run-time.
  - Errors only caught at run-time, when it is too late.
- Key advantages for dynamically typed languages:
  - More flexibility. There are many programs that are safe, but cannot be typed at compile time, *e.g.*,

```
x = 100
x = x*x
println ( x+1 )
x = "hello"
x = concatenate( x, " world" )
println ( x )
```

  - Moreover, vis-a-vis languages like Java, we don't have to write type annotations. Less work . . .

The program provides less predictability, but more flexibility — because it demands less **global consistency**.

Dynamic vs static typing

For large-scale industrial programming, the disadvantages of dynamically typed languages become overwhelming.

Dynamic vs static typing

For large-scale industrial programming, the disadvantages of dynamically typed languages become overwhelming. To compensate, static types are often retro-fitted, *e.g.*:

- **Javascript:** Flow (Facebook) and Typescript (Microsoft)
- **Python:** mypy and PyAnnotate (Dropbox)

Dynamic vs static typing

For large-scale industrial programming, the disadvantages of dynamically typed languages become overwhelming. To compensate, static types are often retro-fitted, *e.g.*:

- **Javascript:** Flow (Facebook) and Typescript (Microsoft)
- **Python:** mypy and PyAnnotate (Dropbox)

But also: we don't have to use a dynamically typed language to get away with leaving out type annotations.

- If we're willing to give up the security of redundant specifications, we can often let the compiler **infer** the type information of a program.

Dynamic vs static typing

For large-scale industrial programming, the disadvantages of dynamically typed languages become overwhelming. To compensate, static types are often retro-fitted, *e.g.*:

- **Javascript:** Flow (Facebook) and Typescript (Microsoft)
- **Python:** mypy and PyAnnotate (Dropbox)

But also: we don't have to use a dynamically typed language to get away with leaving out type annotations.

- If we're willing to give up the security of redundant specifications, we can often let the compiler **infer** the type information of a program.

— The benefits and the constraints of static type-checking can be negotiated in practise

Type-checking vs type-inference

Using type-checking (*e.g.*, in Java), we verify that the programmer-written type-annotations are consistent with the program code.

Type-checking vs type-inference

Using type-checking (*e.g.*, in Java), we verify that the programmer-written type-annotations are consistent with the program code.

Example. We can see that the following is inconsistent:

```
int f ( String x ) {
  if ( x == "Moon" ) return true;
  else return false; }
```

Type-checking vs type-inference

Using type-checking (*e.g.*, in Java), we verify that the programmer-written type-annotations are consistent with the program code.

Example. We can see that the following is inconsistent:

```
int f ( String x ) {
  if ( x == "Moon" ) return true;
  else return false; }
```

But... in the meantime, time has been wasted writing a type specification that even a computer could figure out from context.

Type-checking vs type-inference

Using type-checking (*e.g.*, in Java), we verify that the programmer-written type-annotations are consistent with the program code.

Example. We can see that the following is inconsistent:

```
int f ( String x ) {
  if ( x == "Moon" ) return true;
  else return false; }
```

But... in the meantime, time has been wasted writing a type specification that even a computer could figure out from context.

Type-inference is exactly the approach of having the compiler determine which types are consistent with the program (sometimes with a *little* help from the programmer).

- Rather than check for conformity to a supplemental specification, we require *internal consistency* of code, with respect to *possible* type specifications.

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, ...).

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, . . .).

Example #1.

```
fun f ( x : ??? , y : ??? ) : ???
  =  if (x = y) then y else x+1;
```

What types could you give to the arguments and the return value of `f` ?

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, ...).

Example #1.

```
fun f ( x : ??? , y : ??? ) : ???
  =  if (x = y) then y else x+1;
```

What types could you give to the arguments and the return value of `f` ?

- `x : int`, `y : int`, `f : int * int -> int`.

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, ...).

Example #1.

```
fun f ( x : ??? , y : ??? ) : ???
  =  if (x = y) then y else x+1;
```

What types could you give to the arguments and the return value of `f`?

- `x : int`, `y : int`, `f : int * int -> int`.

Example #2.

```
fun f ( x : ??? , y : ??? ) : ???
  = if (x orelse ceil(y) < 1)
    then print("hi") else print("bye");
```

What types could you give to `x`, `y`, and `f`?

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, ...).

Example #1.

```
fun f ( x : ??? , y : ??? ) : ???
  =  if (x = y) then y else x+1;
```

What types could you give to the arguments and the return value of `f`?

- `x : int`, `y : int`, `f : int * int -> int`.

Example #2.

```
fun f ( x : ??? , y : ??? ) : ???
  = if (x orelse ceil(y) < 1)
    then print("hi") else print("bye");
```

What types could you give to `x`, `y`, and `f`?

- `x : bool` and `y : float`

Type-checking vs type-inference

Many functional programming languages implement type-inference (e.g. Haskell, O'Caml, Scala, Rust, . . .).

Example #1.

```
fun f ( x : ??? , y : ??? ) : ???
  =  if (x = y) then y else x+1;
```

What types could you give to the arguments and the return value of `f` ?

- `x : int`, `y : int`, `f : int * int -> int`.

Example #2.

```
fun f ( x : ??? , y : ??? ) : ???
  = if (x orelse ceil(y) < 1)
    then print("hi") else print("bye");
```

What types could you give to `x`, `y`, and `f`?

- `x : bool` and `y : float`
- `f : bool * int -> unit`, where `print : string -> unit`.

Polymorphism (not exam relevant)

What about: `fun f ( x : ??? , y : ??? ) : ??? = y;`

Polymorphism (not exam relevant)

What about: `fun f ( x : ??? , y : ??? ) : ??? = y;`

We can use **any** types `S` and `T` for the two arguments:

```
fun f ( x : S , y : T ) : T = y;
```

This is called **(parametric) polymorphism**.

Polymorphism (not exam relevant)

What about: `fun f ( x : ??? , y : ??? ) : ??? = y;`

We can use **any** types `S` and `T` for the two arguments:

```
fun f ( x : S , y : T ) : T = y;
```

This is called **(parametric) polymorphism**.

But which concrete type `T` should we use? We want to be able to invoke both `f(17,20)` **and** `f(1.7,true)`, without further qualifying the function `f`. Any concrete choice of `T` would prevent this.

Polymorphism (not exam relevant)

What about: `fun f ( x : ??? , y : ??? ) : ??? = y;`

We can use **any** types `S` and `T` for the two arguments:

```
fun f ( x : S , y : T ) : T = y;
```

This is called **(parametric) polymorphism**.

But which concrete type `T` should we use? We want to be able to invoke both `f(17,20)` **and** `f(1.7,true)`, without further qualifying the function `f`. Any concrete choice of `T` would prevent this.

To deal with this we could use **type variables** which can be instantiated with any type.

Polymorphism (not exam relevant)

What about: `fun f ( x : ??? , y : ??? ) : ??? = y;`

We can use **any** types `S` and `T` for the two arguments:

```
fun f ( x : S , y : T ) : T = y;
```

This is called **(parametric) polymorphism**.

But which concrete type `T` should we use? We want to be able to invoke both `f(17,20)` **and** `f(1.7,true)`, without further qualifying the function `f`. Any concrete choice of `T` would prevent this.

To deal with this we could use **type variables** which can be instantiated with any type. In Java, the concept of **generics** captures this polymorphism, *e.g.*:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();}
```