Why study compiler construction?

Few professionals design and write compilers.

So why teach how to make compilers?

- A good software/telecom engineer understands the high-level **languages** as well as the **hardware**.
  A compiler links these two aspects.
- That is why understanding the compiling techniques is understanding the interaction between the programming languages and the computers.
- Many applications embed small languages for configuration purposes or make their control versatile (think of macros, scripts, data description etc.)

Why study compiler construction? (cont)

The techniques of compilation are necessary for implementing such languages.

Data formats are also formal languages (languages to specify data), like HTML, XML, ASN.1 etc.

The compiling techniques are mandatory for reading, treating and writing data but also to port (migrate) applications (re-engineering). This is a common task in companies.

Anyway, compilers are excellent examples of complex software systems

- which can be rigorously specified,
- which only can be implemented by combining theory and practice.

Function of a compiler

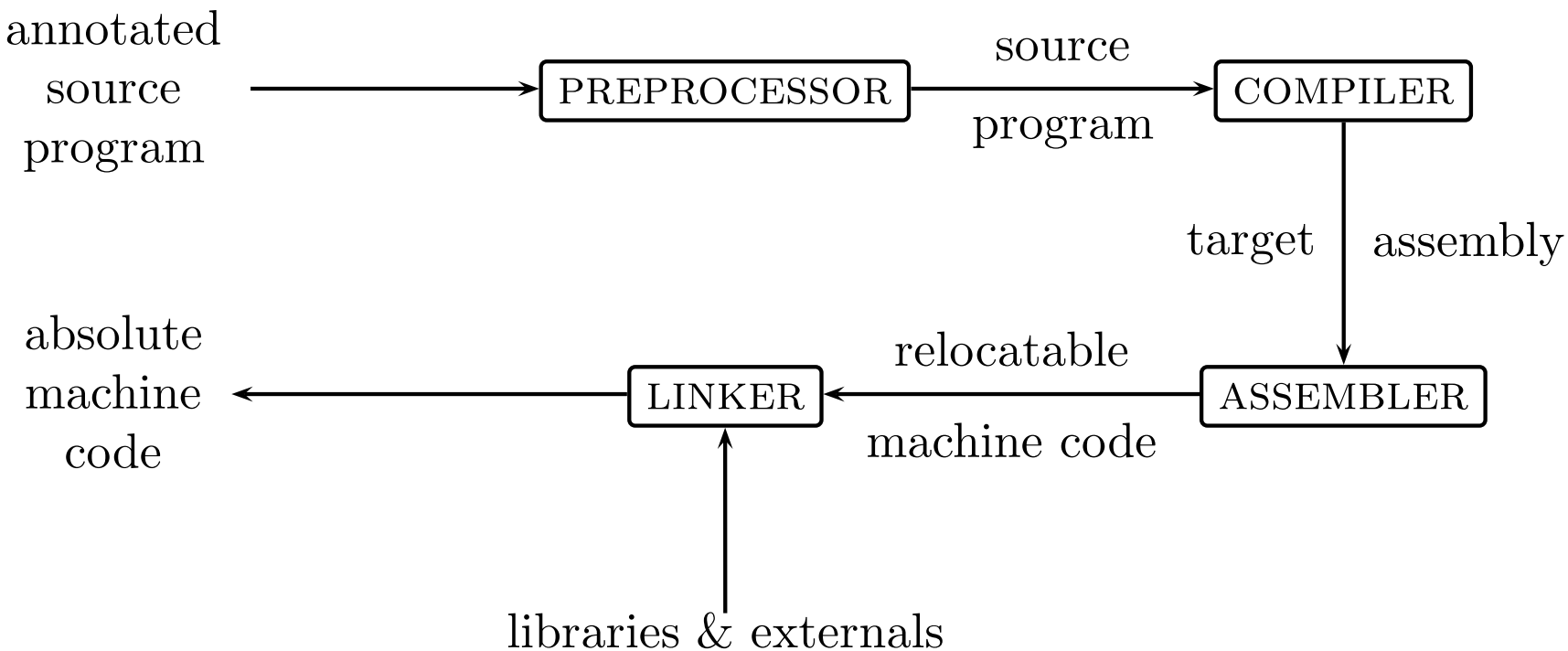
The function of a compiler is to **translate** texts written in a **source language** into texts written in a **target language**.

Usually, the source language is a **programming language**, and the corresponding texts are **programs**. The target language is often an **assembly language**, i.e. a language closer to the machine language (it is the language understood by the processor) than the source language.

Some programming languages are compiled into a **byte-code language** instead of assembly. Byte-code is usually not close to any assembly language. Byte-code is **interpreted** by another program, called **virtual machine** (**VM**), instead of being translated to machine language (which is directly executed by the machine processor): the VM processes the instructions of the byte-code.

Compilation chain

From an engineering point of view, the compiler is one link in a chain of tools:



Compilation chain (cont)

Let us consider the example of the **C language**. A famous free compiler is GNU GCC.

In reality, GCC includes the complete compilation chain, not just a C compiler:

- to only preprocess the sources: `gcc -E prog.c` (standard output)
  Annotations are introduced by #, like `#define x 6`
- to preprocess and compile: `gcc -S prog.c` (output `prog.s`)
- to preprocess, compile and assemble: `gcc -c prog.c` (output `prog.o`)
- to preprocess, compile, assemble and link: `gcc -o prog prog.c` (output `prog`)
  Linkage can be directly called using `ld`.

The analysis-synthesis model of compilation

In this class we shall detail only the compilation stage itself.

There are two parts to compilation: **analysis** and **synthesis**.

1. The analysis part breaks up the source program into constituent pieces of an **intermediary representation** of the program.
2. The synthesis part constructs the target program from this intermediary representation.

In this class we shall restrict ourselves to the analysis part.

Analysis

The analysis can itself be divided into three successive stages:

1. **linear analysis,** in which the stream of characters making up the source program is read and grouped into **lexemes** that are sequences of characters having a collective meaning; sets of lexemes with a common interpretation are called **tokens**;
2. **hierarchical analysis,** in which tokens are grouped hierarchically into nested collections (**trees**) with a collective meaning;
3. **semantic analysis,** in which certain checks are performed to ensure the components of a program fit together meaningfully.

In this class we shall focus on linear and hierarchical analysis.

Lexical analysis

In a compiler, linear analysis is called **lexical analysis** or **scanning**.

During lexical analysis, the characters in the assignment statement

```
position := initial+rate*60
```

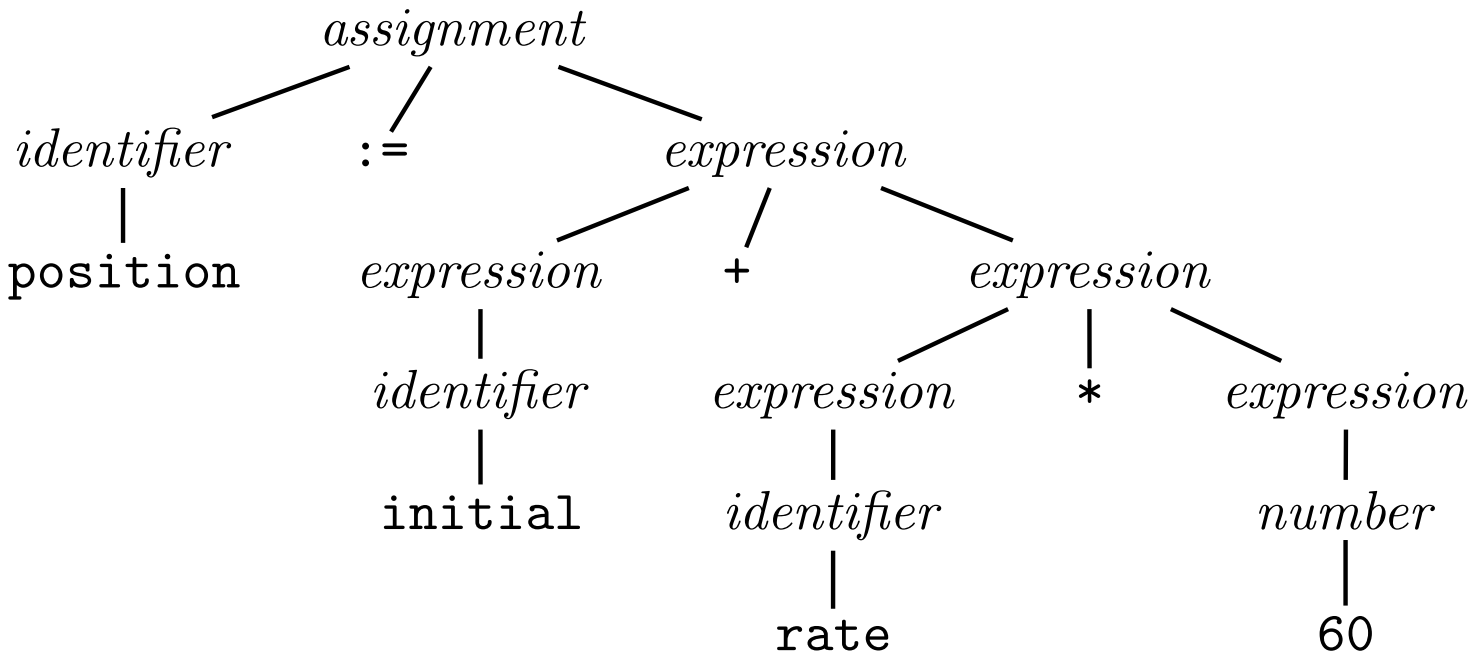
would be grouped into the following lexemes and tokens (see facing table).

The blanks separating the characters of these tokens are normally eliminated.

Token	Lexeme
identifier	`position`
assignment symbol	`:=`
identifier	`initial`
plus sign	`+`
identifier	`rate`
multiplication sign	`*`
number	60

Syntax analysis

Hierarchical analysis is called **parsing** or **syntax analysis**. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source are represented by a **parse tree** such as:



Syntax analysis (cont)

In the expression

```
initial + rate * 60
```

the phrase

```
rate * 60
```

is a logical unit because the usual conventions of arithmetic expressions tell us that multiplication is performed prior to addition.

Thus, because the expression

```
initial + rate
```

is followed by a `*`, it is **not** grouped into the same subtree.

Syntax analysis (cont)

The hierarchical structure of a program is usually expressed by **recursive rules**. For instance, an expression can be defined by a set of cases:

1. Any *identifier* is an expression.
2. Any *number* is an expression.
3. If $expression_1$ and $expression_2$ are expressions, then so are
   - 3.1 $expression_1$ + $expression_2$
   - 3.2 $expression_1$ * $expression_2$
   - 3.3 ($expression_1$)

Syntax analysis (cont)

Rule 1 and 2 are non-recursive base rules, while the others define expressions in terms of operators applied to other expressions.

`initial` and `rate` are identifiers.

Therefore, by rule 1, `initial` and `rate` are expressions.

60 is a number.

Thus, by rule 2, we infer that 60 is an expression.

Then, by rule 2, we infer that `rate * 60` is an expression.

Thus, by rule 1, we conclude that `initial + rate * 60` is an expression

Syntax analysis (cont)

Similarly, many programming languages define statements recursively by rules such as

1. If *identifier* is an identifier and *expression* is an expression, then

   identifier `:=` *expression*

   is a statement.

2. If *expression* is an expression and *statement* is a statement, then

   `while (`*expression*`) do` *statement*

   `if (`*expression*`) then` *statement*

   are statements.

Syntax analysis (cont)

The division between lexical and syntactic analysis is somewhat arbitrary.

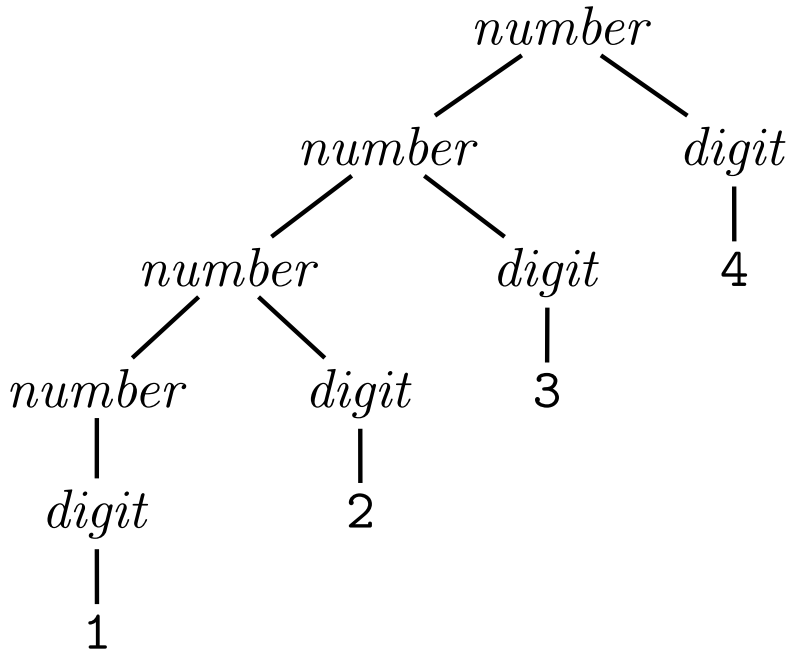
For instance, we could define the integer numbers by means of recursive rules:

1. a *digit* is a *number* (base rule),
2. a *number* followed by a *digit* is a *number* (recursive rule).

Imagine now that the lexer does **not** recognise numbers, just digits. The parser therefore uses the previous recursive rules to group in a parse tree the digits which form a number.

Syntax analysis (cont)

For instance, the parse tree for the number 1234, following these rules, would be



But notice how this tree actually is almost a list.

The structure, i.e. the embedding of trees, is indeed not meaningful here.

For example, there is no obvious meaning to the separation of 12 (same subtree at the leftmost part) in the number 1234.

Syntax analysis (cont)

Therefore, pragmatically, the best division between the lexer and the parser is the one that simplifies the overall task of analysis.

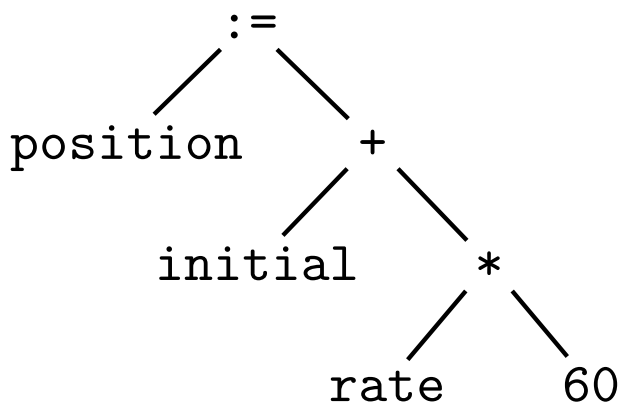
One factor in determining the division is whether a source language construct is inherently recursive or not: lexical constructs do not require recursion, while syntactic construct often do.

For example, recursion is not necessary to recognise identifiers, which are typically strings of letters and digits beginning with a letter: we can read the input stream until a character that is neither digit nor letter is found, then these read characters are grouped into an identifier token.

On the other hand, this kind of linear scan is not powerful enough to analyse expressions or statements, like matching parentheses in expressions or { and } in block statements: a nesting structure is needed.

Syntax analysis (cont)

The parse tree page 10 describes the syntactic structure of the input. A more common *internal* representation of this syntactic structure is given by



An **abstract syntax tree** (or just **syntax tree**) is a compressed version of the parse tree, where only the most important elements are retained for the semantic analysis.

Semantic analysis

The semantic analysis checks the syntax tree for meaningless constructs and completes it for the synthesis.

An important part of semantic analysis is devoted to **type checking**, i.e. checking properties on how the data in the program is combined.

For instance, many programming languages require an error to be issued if an array is indexed with a floating-point number (called *float*).

Some languages allow such floats and integers to be mixed in arithmetic expressions. Some do not (because internal representation of integers and floats is very different, as well as the cost of the corresponding arithmetic functions).

Semantic analysis (cont)

In our example, page 18, assume all identifiers were declared as floats.

The type-checking compares the type of `rate`, which is float, and of 60, which is integer. Let us assume our language allows these two types of operands for the multiplication $*$.

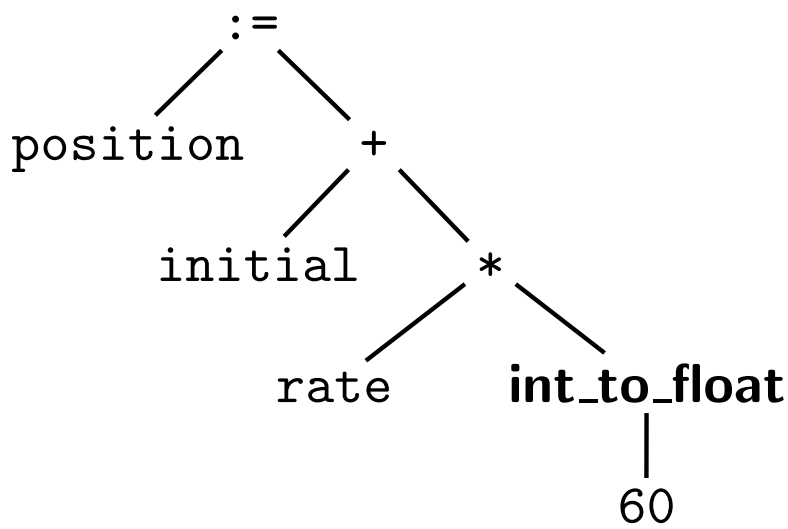
Then the analyser must insert a special node in the syntax tree which represents a **type cast** from integer to float for 60.

At the level of the programming language, a type cast is the identity function (in mathematics: $x \mapsto x$), so the value is not changed, but the type of the result is different from the type of the argument.

This way the synthesis will know that the assembly code for such a conversion has to be output.

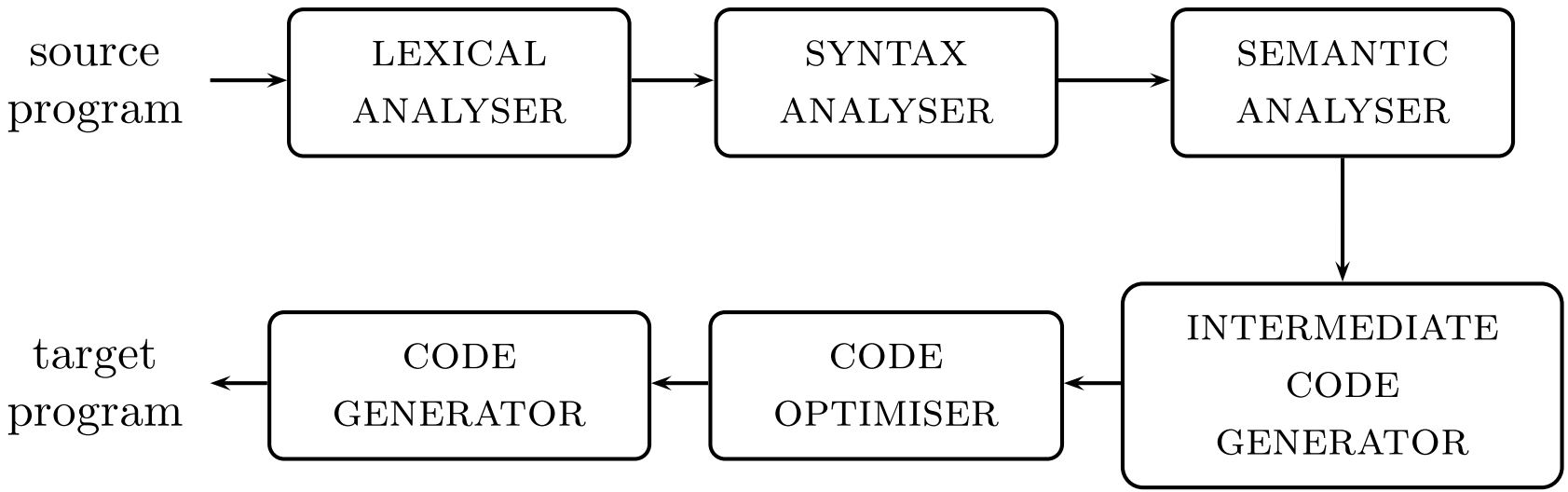
Semantic analysis (cont)

Hence the semantic analysis issues no error and produces the following **annotated syntax tree** for the synthesis:



Phases

Conceptually, a compiler operates in **phases**, each of which transforms the program from one representation to another. A typical decomposition of a compiler is as follows:



The first row makes up the analysis and the second the synthesis.

Phases/Symbol table

The previous figure did not depict another component which is connected to all the phases: the **symbol table manager**. A symbol table is a two-column table whose first column contains identifiers collected in the program and the second column contains any interesting information, called **attributes**, about their corresponding identifier. Example of identifier attributes are

- the allocated storage,
- the type,
- the **scope** (i.e. where in the program it is valid),
- in case of procedures names, the number and type of the parameters, the method of passing each argument (e.g., by reference) and the result type, if any.

Phases/Symbol table (cont)

When an identifier in the source program is detected by the lexical analyser (or simply called **lexer**), this identifier is entered into the symbol table.

However, some attributes of an identifier cannot normally be determined during lexical analysis (or simply called **lexing**). For example, in a Pascal declaration like

```
var position, initial, rate: real;
```

the type `real` is not known when `position`, `initial` and `rate` are recognised by the lexical analyser.

The remaining phases enter information about the identifiers into the symbol table and use this information. For example, the semantic analyser needs to know the type of the identifiers to generate intermediate code.

Phases/Error detection and reporting

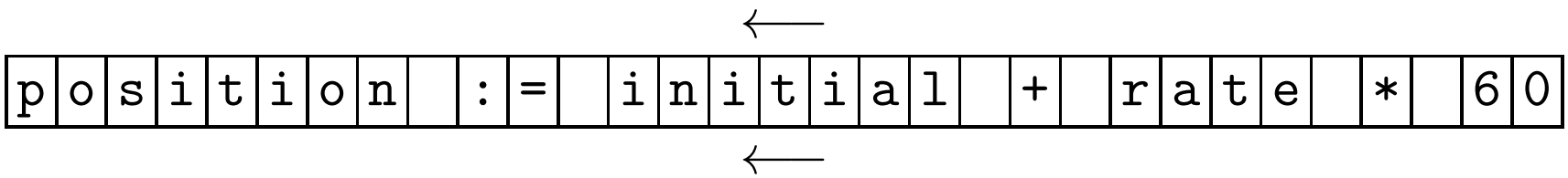
Another compiler component that was omitted from picture page 22 because it is also connected to all the phases is the **error handler**.

Indeed, each phase can encounter errors, so each phase must somehow deal with these errors. Here come some examples.

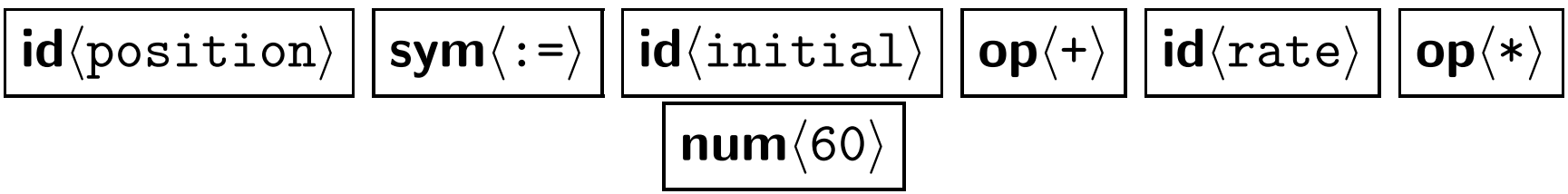
- Lexical analysis finds an error if a series of characters do not form a token.
- Syntax analysis finds an error if the relative position of a group of tokens is not described by the grammar (syntax).
- Semantic analysis finds an error if the program contains the addition a an integer and an array.

Phases/The analysis phase/Lexing

Let us consider again the analysis phase and its sub-phases in more details, following a previous example. Consider the next character string



First, as we stated page 9, the lexical analysis recognises the tokens of this character string (which can be stored in a file). The output of the lexing is a stream of tokens like



where **id** (*identifier*), **sym** (*symbol*), **op** (*operator*) and **num** (*number*) are the token names and between brackets are the **lexemes**.

Phases/The analysis phase/Lexing (cont)

The lexer also outputs or updates a symbol table like[1]

Identifier	Attributes
`position`	...
`initial`	...
`rate`	...

The attributes often include the position of the corresponding identifier in the original string, like the position of the first character either counting from the start of the string or through the line and column numbers.

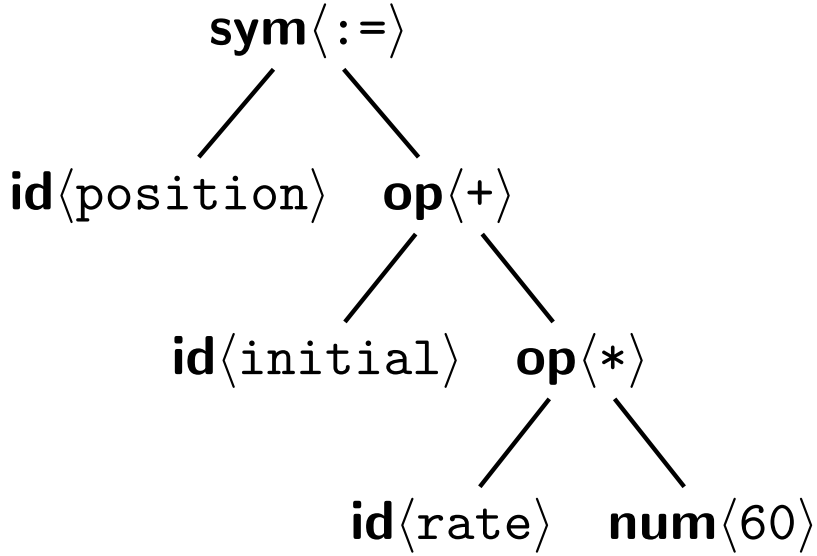
[1] Even if the table is named "symbol table" it actually contains information about identifiers only.

Phases/The analysis phase/Parsing

The parser takes this token stream and outputs the corresponding syntax tree and/or report errors.

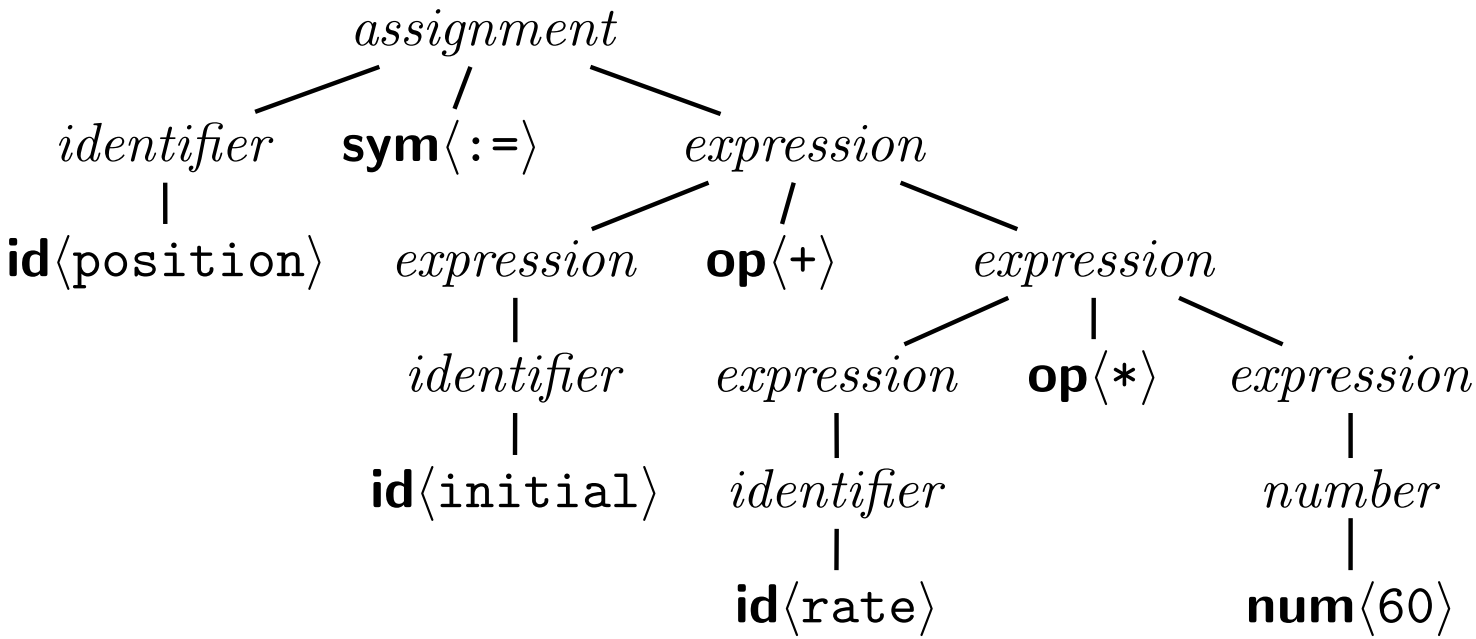
Page 18, we gave a simplified version of this syntax tree. A refined version is given in the facing column.

Also, if the language requires variable definitions, the syntax analyser can complete the symbol table with the type of the identifiers.



Phases/The analysis phase/Parsing (cont)

The parse tree can be considered as a **trace** of the syntax analysis process: it summarises all the recognition work done by the parser. It depends on the syntax rules (i.e. the grammar) and the input stream of tokens.



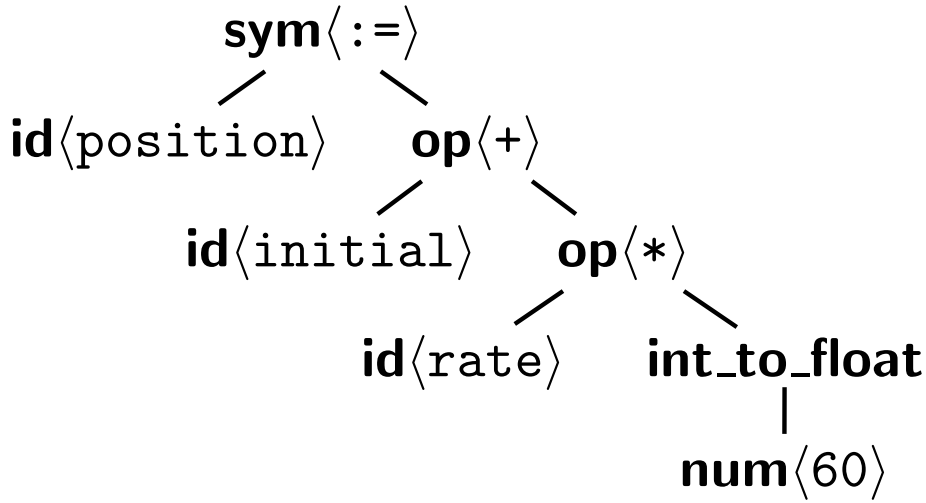
Phases/The analysis phase/Semantic analysis

The semantic analysis considers the syntax tree and checks certain properties depending on the language, typically it makes sure that the valid syntactic constructs also have a certain meaning (with respect to the rules of the language).

We saw page 21 that this phase can annotate or even add nodes to the syntax tree. It can as well update the symbol table with the information newly gathered in order to facilitate the code generation and/or optimisation.

Phases/The analysis phase/Semantic analysis (cont)

Assuming that our toy language accepts that an integer is mixed with floats in arithmetic operations, the semantic analysis can insert a type cast node. A new version of the annotated syntax tree would be:



Note that the new node is not a token, just a (semantic) tag for the code generator.

Phases/The synthesis phase

The purpose of the synthesis phase is to use all the information gathered by the analysis phase in order to produce the code in the target language.

Given the annotated syntax tree and the symbol table, the first sub-phase consists in producing a program in some artificial, intermediary, language.

Such a language should be independent of the target language, while containing features common to the *family* the target language belongs to.

For instance, if the target language is the PowerPC G4 microprocessor, the intermediary language should be like an assembly of the IBM RISC family.

Phases/The synthesis phase (cont)

If we want **to port a compiler** from one platform to another, i.e., make it generate code for a different OS or processor, such intermediary language comes handy: if the new platform share some features with the first one, we only have to rewrite the code generator component of the compiler — not the whole compiler.

It may be interesting to have the same intermediary language for different source languages, allowing the sharing of the synthesis.

We can think of an intermediary language as an assembly for an **abstract machine** (or processor). For instance, our example could lead to the code

```
temp1 := inttoreal(60)
temp2 := id_rate * temp1
temp3 := id_initial + temp2
id_position := temp3
```

Phases/The synthesis phase (cont)

Another point of view is to consider the intermediary code as a tiny subset of the source language, as it retains some high-level features from it, like, in our example, variables (instead of explicit storage information, like memory addresses or register numbers), operator names etc.

This point of view enables optimisations that otherwise would be harder to achieve (because too many aspects would depend closely on many details of the target architecture).

Phases/The synthesis phase (cont)

This kind of assembly is called **three-address code**. It has several properties:

- each instruction has at most one operator (in addition to the assignment);
- each instruction can have at most three operands;
- some instructions can have less than three operands (e.g. the first and last instruction);
- the result of an operation must be linked to a variable;

As a consequence, the compiler must order well the code for the sub-expressions, e.g. the second instruction must come before the third one because the multiplication has priority on addition.

Phases/The synthesis phase/Code optimisation

The code optimisation phase attempts to improve the intermediate code, so that faster-running target code will result.

The code optimisation produces intermediate code: the output language is the same as the input language.

For instance, this phase would find out that our little program would be more efficient this way:

```
temp1 := id_rate * 60.0
id_position := id_initial + temp1
```

This simple optimisation is based on the fact that type casting can be performed at compile-time instead of run-time, but it would be an unnecessary concern to integrate it in the code generation phase.

Phases/The synthesis phase/Code generation

The code generation is the last phase of a compiler. It consists in the generation of target code, usually relocatable assembly code, from the optimised intermediate code.

A crucial aspect is the assignment of variables to registers.

For example, the translation of code page 36 could be

```
MOVF id_rate, R2
MULF #60.0, R2
MOVF id_initial, R1
ADDF R2, R1
MOVF R1, id_position
```

The first and second operands specify respectively a source and a destination.

The `F` in each instruction tells us that the instruction is dealing with floating-point numbers.

Phases/The synthesis phase/Code generation (cont)

This code moves the contents of the address `id_rate` into register 2, then multiplies it with the float 60.0.

The # signifies that `60.0` is a constant.

The third instruction moves `id_initial` into register 1 and adds to it the value previously computed in register 2.

Finally, the value in register 1 is moved to the address of `id_position`.

Implementation of phases into passes

An implementation of the analysis is called a **front-end** and an implementation of the synthesis **back-end**.

A **pass** consists in reading an input file and writing an output file.

It is possible to group several phases into one pass in order to interleave their activity.

- On one hand, this can lead to a greater efficiency since interactions with the file system are much slower than with internal memory.
- On the other hand, this architecture leads to a greater complexity of the compiler — something the software engineer always fears.

Implementation of phases into passes (cont)

Sometimes it is difficult to group different phases into one pass.

For example, the interface between the lexer and the parser is often a single token. There is not a lot of activity to interleave: the parser requests a token to the lexer which computes it and gives it back to the parser. In the meantime, the parser had to wait.

Similarly, it is difficult to generate the target code if the intermediate code is not fully generated first. Indeed, some languages allow the use of variables without a prior declaration, so we cannot generate immediately the target code because this requires the knowledge of the variable type.