Part I : Overview and Foundations

- Why study compiler construction ?
- The task and structure of a compiler.
- Language and Syntax.
- Formal Languages.

Why Study Compiler Construction ?

There are very few people writing compilers for a living.

So why bother learning about compilers ?

- A competent computer professional knows about high-level programming and about hardware
- A compiler connects the two.
- Therefore, understanding compilation techniques is essential for understanding how programming languages and computers hang together.
- Many applications contain little languages for customization and flexible control
  - Examples : Word macros, scripts for graphics & animation, data layout descriptions.

Why Study Compiler Construction ? (2)

- Compiler techniques are needed to properly design and implement these extension languages.
- Data formats are also formal languages. More and more data in interchangeable format look like a formal language text (e.g. HTML, XML).
- Compiler techniques are useful for reading, manipulating and writing data.
- Besides, compilers are excellent examples of large and complex system
  - which can be specified rigorously,
  - which can be implemented only by combining theory and practise.

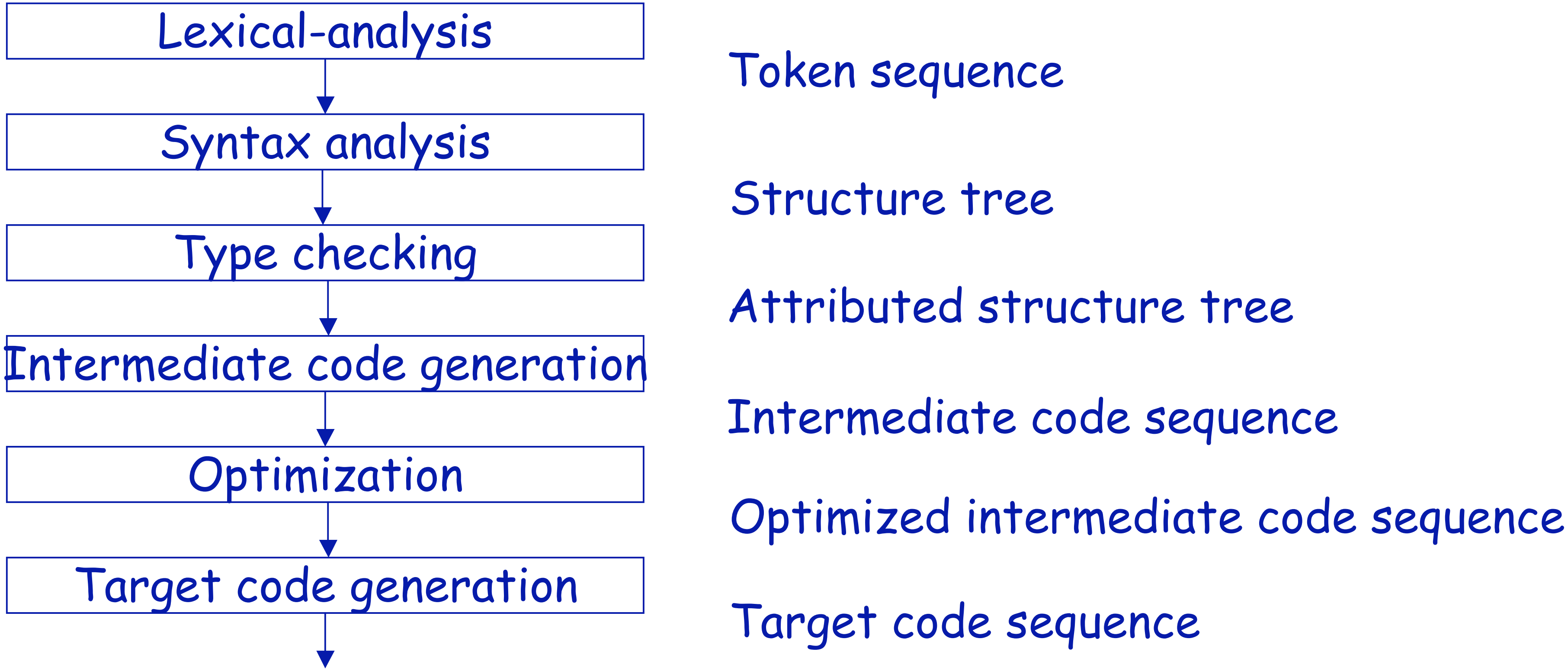
The Task of a Compiler

- The main task of a compiler is to map programs written in a given *source* language into a *target* language.
- Often, the source language is a programming language and the target language is a machine language.
- Some Exceptions : Source-to-source translators, machine-code translation, data manipulation in XML.
- Part of the task of a compiler is also to detect whether a given program conforms to the rules of the source language.
- A specification of a compiler consists of
  - A specification of its source- and target languages,
  - A specification of a mapping between them.

Languages

- Formally, a language is a set of *strings* (sentences).
- In practice, each string in a language has a *structure* which can be described by a *tree.*
- Structure rules for sentences are defined by a *grammar*.
- Example :
  - The sentences of a programming language are (legal) programs.
  - Programs are sentences of words (or : *symbols, tokens*); their structure is given by a context-free grammar.
  - Words themselves are sequences of characters; the structure of which can also be given by a grammar.

Compiler-Structure



- Phases are not necessarily executed one after the other.
- Intermediate data structures do not always exist in their entirety at any one time.

Language and Syntax

- Language has structure which is determined by a grammar.
- **Example**: A correct sentence consists of a subject, followed by a verb.
- This can be expressed by the grammar:

```
Sentence = Subject Verb
```

- Let's complete this with two more productions:

```
Subject = "Peter" | "Chelsea"
Verb = "runs" | "stops"
```

- Then this defines 4 possible sentences

  Peter runs | Peter stops | Chelsea runs | Chelsea stops

- Usually, languages contain an infinite number of sentences.

Language and Syntax (2)

- An infinite number of sentences can be expressed by a finite number of productions by using recursion over some symbols.
- Example :

```
Number = Digit   |   Digit Number
Digit    = "0"  | "1" | "2" | "3" | "4" | . . . | "9".
```

Generates :

0
12
123
1024

etc.

Formal Languages

A language is formally defined by :

•A set of *terminal symbols.*

•A set of *non-terminal symbols*

•A set of *syntactic rules* (or : *productions*)

•A *start symbol.*

A grammar defines as its language the set of those sequences of terminal symbols which can be derived from the start symbol by successive application of productions.

The language of (context-free) grammars

```
syntax      = production syntax | (empty)
production  = identifier "=" expression "."
expression  = term | expression "|" term
term        = factor | term factor | "(empty)"
factor      = identifier | string
identifier  = letter | identifier letter | identifier digit
string      = "\"" stringchars "\""
stringchars = stringchars stringchar | (empty)
stringchar  = escapechar | plainchar
escapechar  = "\\" char
plainchar   = charNoQuote
char        = «any printable character».
charNoQuote = «any printable character except `"'.»
```

The Language of (context-free) Grammars (2)

- This was originally developed by J. Backus and P. Naur for the definition of Algol 60.
- That 's why it 's commonly called *Backus-Naur form*, or *BNF.*

- Exercise : Determine startsymbol, terminal symbols and nonterminals for this grammar.

Extended Backus Naur Form

Grammars can often be simplified and shortened by using two more constructs :

- {x} expresses *repetition:* zero, one or more occurrences of x.
- [x] expresses *option:* zero or one occurrences of x.

The resulting formalism is called *extended Backus-Naur form,* or *EBNF.* It 's syntax is:

Extended Backus Naur Form (2)

```
syntax       =  {production}
production   =  identifier "=" expression "."
expression   =  term {"|" term}
term         =  {factor}
factor       =  identifier
             |  string
             |  "(" expression ")"
             |  "[" expression "]"
             |  "{" expression "}"
identifier   =  letter { letter | digit }
string       =  "\"" {stringchar} "\"
```

(rest as for BNF)

Exercise : Write the grammar for (possibly signed) integer numbers in - BNF, - EBNF.