1 Part I: Compilation: Overview and Foundations

- Why study compilation?
- The task and structure of a compiler
- Language and syntax
- Formal languages

2 Why study Compiler Construction?

There are very few people who will write compilers for a living, so why bother?

- A competent computer professional knows about high-level programming and 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
  - Word macros, layout descriptions, document descriptions
- Compiler techniques are needed to properly design and implement these extension languages

3 Why study Compiler Construction? (2)

- Data formats are also formal languages. More and more data in interchangeable format look like a formal language (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 practice

4 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

5 The Task of an Interpreter

- The task of an interpreter is to map programs written in a given *source* language into an *internal representation* and then to *execute* the internal representation.
- Some languages (LISP, SCHEME, BASIC, Smalltalk, PROLOG) are mostly interpreted.
- Some languages (Java, Pascal, PROLOG) are compiled into *abstract machine code*, which is then interpreted by a *virtual machine*.
- Advantage of compilation:
  - execution speed
- Advantage of interpretation:
  - quick turn-around
  - portability

6 Compiler-Structure

Lexical analysis	⇒ *Token sequence*
Syntax analysis	⇒ *Structure tree*
Semantic analysis	⇒ *Attributed structure tree*
Intermediate code generation	⇒ *Intermediate code sequence*
Optimization	⇒ *Intermediate code sequence*
Target code generation	⇒ *Target code sequence*

- Phases are not necessarily executed one after another.
- Intermediate data structures do not always exist in their entirety at any one time.
- In the case of an interpreter, interpretation can happen on the attributed syntax tree or on the intermediate code. For simple languages somtimes even during parsing instead of building a tree.

7 Languages

- Formally, a language is a set of flat *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 *tokens* (words). The structure of a program is given by a context-free grammar.
  - Words themselves are sequences of characters, the structure of words can also be given by a grammar.

8 Language and Grammars

- A 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.

Q: Write a grammar for integer numbers!

9 Language and Grammars (2)

- An infinite number of sentences can be expresses by a finite number of productions by using recursion over some symbols.
- Example:
  Number = Digit | Digit Number.
  Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
- allows
  0 | 12 | 347 | 0013 | ...

10 Context-free Grammars

A context-free grammar is formally defined by

- A set of *terminal symbols* ("0", "7", "Chelsea")
- A set of *non-terminal symbols* (Subject)
- A set of *syntactic rules* (or: *productions*) (Subject="Chelsea"|"Peter".)
- A *start* symbol (Sentence)

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.

Q: What are all the terminals, non-terminals, rules, start-symbols of the number example?

11 BNF (Backus-Naur Form)

This was originally developed by J.Backus and P.Naur for Algol 60.

- a production (or rule) consists of a left-hand-side and a right-hand-side.
- The left-hand-side is a single non-terminal.
- The right-hand-side contains terminals and non-terminals, we use
  - We use | for alternatives.
  - We use juxtaposition for concatenation.
  - concatenation binds stronger than |.
- We often use quotes for terminals.
- We will usually distinguish terminals which are not quoted and non-terminals by capitalization.

12 EBNF (Extended BNF)

- We use (...) for grouping.
- We use ϵ for the empty word.
- We use [E] to stand for (ϵ | E)
- We use { E } to stand for (ϵ | E | EE | EEE | ...)

We can now write
Number = Digit { Digit }.
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
or
Sentence = ("Peter" | "Chelsea") ("runs" | "stops").

13 Two Level Description

- Context-free syntax of arithmetic expressions

```
Expression = Expression ( minus | plus ) Term | Term.
Term       = Term ( times | div ) Factor | Factor.
Factor     = numlit | lparen Expression rparen.
```

- Lexical syntax of arithmetic expressions

```
times  = "*".
div    = "/".
plus   = "+".
minus  = "–".
lparen = "(".
rparen = ")".
numlit = digit { digit }.
digit  = "0" | ... | "9".
```


14 Two Level Description (2)

Why two levels

- White space, comments
- Structure
- We think that way (sentence, word, character).

For a practical specification we will use:

- Context-free Syntax

```
Expression = Expression ( "–" | "+" ) Term | Term.
Term       = Term ( "*" | "/" ) Factor | Factor.
Factor     = numlit | "(" Expression ")".
```

- Lexical Syntax

```
numlit = digit { digit }.
digit  = "0" | ... | "9".
```

But for the actual implementation we will use the first scheme.

15 Exercises

- How many terminals, non-terminals, productions, start-symbols does the context-free Expression-grammar have?
- Give a grammar for floating point numbers of the form 123.45.
- Can you even extend it to a description which includes 12.4e17 and 0.23e–24.