1 Part I: Compilation: Overview and Foundations

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

2 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.

3 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
- Virtual machines have a bit of both.

4 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.

5 Why study Compiler Construction?

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

- Many programs have to read and analyze input.
  - parameter-files
  - user-commands
  - XML
- Analyzing binary data is very similar to analyzing source programs
- How to organize analyzed information, how to manipulate and how to output it.
  - pretty printer

6 Why study Compiler Construction(2)?

- Understanding compilers means understanding programming languages better.
  - Designing small languages (user commands)
- Connects software and hardware?
- Connects theory and practice?

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, Sentence)
- A set of *syntactic rules* (*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.

- A language is a set of sentences.
- A grammar is one description of a language.
- There are in general many grammars for a language.

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.
  - terminals never occur on left-hand-sides
- The right-hand-side contains terminals and non-terminals, we use
  - We use | for alternatives.
  - We use juxtaposition for concatenation.
  - concatenation binds stronger than |.
    A ::= b c | d means A ::= (b c) | d and not A ::= b (c | d)
- We often use quotes or all capitals for terminals.

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").

Q: Can I replace all recursion by {}? Q: Can I replace all {} by recursion?

13 No/Yes

We cannot write this without recursion:
Par ::= "(" Par ")" | "3"

We can transform every grammar in EBNF into a grammar in BNF, that describes the same language (later).

Q: What is the difference between the above and

Par ::= (Par) | "3"

14 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".

  White space consist of " ", "\t", "\n".

15 Two Level Description (2)

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.

- Tokens like NUMLIT are terminals in the context-free syntax
- But they are non-terminals in the lexical syntax.

16 Two Level Description (2)

Why two levels?

- We think that way (sentence, word, character).
- White space, comments are dealt with in one place.
- Efficiency (Splitting in Scanner and Parser).