Compilation A

Erik Stenman, LAMP, INR 315
Erik.stenman@epfl.ch

Nikolay Mihaylov, LAMP, INR 329
Nikolay.Mihaylov@epfl.ch

Content

1. Compiler Overview
2. Lexical Analysis
3. Syntactic Analysis
4. Abstract Syntax
5. Semantic Analysis
6. Code generation
7. Interpretation

Schedule

- Fridays 08-09 CO2 – Lecture.
- Fridays 11-12 INF1 – Practical work.
- Sometimes, like today, both time slots will be used for lectures in CO2. The exact schedule is available on the web.

Course Material

- Course page:
  http://lampwww.epfl.ch/teaching/CompilationA/2003/html/
- Book:
  - **Modern Compiler Implementation in Java (second edition)**
    by Andrew W. Appel and Jens Palsberg

The Course

- Lectures.
- Exercises – not obligatory.
- Assignment – not obligatory.
- Exam – very hard without doing exercises and the assignment.

The assignment

- Implement a small programming language.
  - Including a compiler and an interpreter for the language.
  - A framework will be given and each week you will have to do one small step towards the full implementation.
- You may work in groups of 2 to 3 persons.
- More information will be given next Friday and on the Web.

Part I

Part I: Compilation

- What is a compiler?
- Why is this interesting to study?
- What is a language and how do we describe languages?

Part I

The Task of a Compiler

- The main task is to map programs written in a *source* language into a *target* language.
- Often the source language is a programming language and the target language is a machine language.
- The same techniques can also be used for source-to-source translators, machine-code translation, data manipulation e.g. in XML.
- Part of the task of a compiler is also to detect whether a given program conforms to the rules of a language.

Part I

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, BASIC) are mostly interpreted.
- Some languages (Java, Erlang, Prolog) are compiled into *abstract machine code*, which is then interpreted by a *virtual machine*.

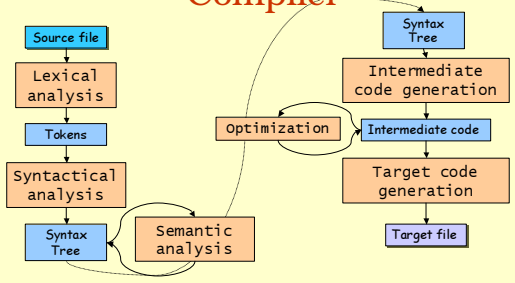
Part I

Native code vs. Interpretation

	Execution speed	Compile time	Portability
Native code	Fast	Long	Hard
Virtual machines	Medium	Medium	Medium
Interpretation	Slow	Short	Easy

Part I

The Structure of a Compiler



Part I

The Structure of a Compiler

- Phases are not necessarily executed after each other.
- Intermediate data structures does not always exist in their entirety.
- In the case of an interpreter, the interpretation can take place on the syntax tree or on the intermediate code. For simple languages sometimes even during parsing.

Part I

Why Study Compiler Construction?

- Many programs have to read and analyze input:
  - Configuration files.
  - User commands.
  - XML, HTML
- Techniques used in a compiler can be used when analyzing other types of data – it teaches you how to organize, manipulate and output information from an analysis.

Part I

Why Study Compiler Construction?

- Understanding of compilers gives better understanding of languages:
  - You will be able to design languages.
  - You will understand how programs are executed and hence will be able to write more efficient programs.
- Compilers connect software and hardware.

Part I

What is a language?

- The first task of a compiler is to understand the meaning of a program written in a programming language.
- Before we can solve this problem we need to have a clear and concise understanding of what a *language* is.

Part I

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 in a tree.
- The Structural rules for sentences are defined by a *grammar*.

Part I

Grammars

- A grammar describes all possible sentences in a language by describing the parts of a legal sentence.
- Example: In a simple version of the English language all sentences consists of a subject, followed by a verb.
- This can be described by the grammar:

```
Sentence ::= Subject " " Verb.
```

Part I

Grammars

- To make the grammar

```
Sentence ::= Subject " " Verb.
```

  complete we need to add two more *productions:*

```
Subject ::= "Erik" | "Nikolay"
Verb ::= "runs" | "stops"
```

This defines a language with 4 sentences:
Erik runs | Erik stops |
Nikolay runs | Nikolay stops

Part I

Grammars

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

```
Number ::= Digit | Digit Number
Digit ::= “0” | “1” | “2” | “3”
        | “4” | “5” | “6”
        | “7” | “8” | “9”
```

Part I

Context-free Grammars

- A context-free grammar is formally defined by:
  - A set of *terminal symbols* (“0”, “Erik”). (The set of all terminal symbols makes up the alphabet of the language.)
  - A set of *non-terminal symbols* (Subject, Sentence).
  - A set of syntactic rules, or *productions* (Subject ::= “Erik” | “Nikolay”).
  - A *start* symbol (Sentence).

Part I

The language of a grammar

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

Part I

BNF

- Backus-Naur Form was developed by J. Backus and P. Naur for Algol 60.
- A production (*rule*) consists of a left-hand-side (lhs) and a right-hand-side (rhs).
- The rhs contains terminals and non-terminals.
  - We use | for alternatives.
  - We use juxtaposition for concatenation.
  - Concatenation binds stronger than |
  - We often quote or capitalize terminals.

Part I

EBNF (Extended BNF)

- EBNF allows us to use:
- ( ) – for grouping.
- ε – for the empty word.
- [E] – for (ε | E).
- { E } – for (ε | E | EE | EEE | …)

Part I

EBNF

We can now write:

```
Number ::= Digit {Digit}
Digit ::= “0” | “1” | “2” | “3”
| “4” | “5” | “6” | “7” | “8” | “9”
```

And:

```
Sentence ::=
(“Erik”|“Nikolay”) “ ” (“runs”|“stops”)
```

Part I

EBNF

- Can I replace all recursion by {}?
- No, e.g.:

```
Par ::= "(" Par ")" | "3"
```

- Can I replace all {} by recursion?
- Yes, every grammar in EBNF can be translated to a grammar in BNF.

Part I

Regular Languages

- A language is *regular* of it can be expressed by a single EBNF rule without recursion.
- Since there is only one, non-recursive rule all symbols on the rhs must be terminal symbols.
- The rhs is called a *regular expression (reg-exp)*.
- Regular languages are interesting because they can be recognized by *finite state machines.*

Part I

Regular Expressions

- A regular language can also be described by a number of EBNF rules without recursion.
- There are alternative ways of writing regular expressions. The most common is:

`E*` for `{E}`.	`[ac]` for `("a"|"c")`.
`E?` for `[E]`.	`[a-d]` for `("a"|"b"|"c"|"d")`.
`E+` for `E{E}`.	`[^0-9]` for anything but a digit.

Part I

Two level descriptions

- The description of a language is often divided into two levels: a lexical (words or *tokens*) and a syntactical (sentences).
- Similarly a compiler recognizes sentences in the language in two steps, with a lexical analysis and syntactic analysis.
- The lexical analysis recognizes words in the language and the syntactical analysis recognizes sentences in the language.

Part I

Two level descriptions

- This division is natural to do, since we also think in words and sentences.
- The rules for words are usually a lot simpler than the rules for sentences. Most words can be described by one regular expression, but the description of sentences often requires recursion (to be concise).
- By splitting the recognition we can deal with white space and comments easily and in one place.

Part I

A Small Language

- Syntax of arithmetic expressions

```
Expr ::= Expr (MINUS | PLUS) Term
       | Term
Term ::= Term (TIMES | DIV) Factor
       | Factor
Factor ::= NUMBER | LPAR Expr RPAR
```

Part I

A Small Language

- Lexical description of *tokens*:

```
TIMES   ::= "*"
DIV     ::= "/"
PLUS    ::= "+"
MINUS   ::= "-"
LPAR    ::= "("
RPAR    ::= ")"
NUMBER  ::= DIGIT {DIGIT}
DIGIT   ::= "0" | … | "9"
```

Part I

A Small Language

- Notice that the syntactical description uses recursion, but that the lexical description does not.
- Notice that NUMBER is a terminal in the syntactical description but a non-terminal in the lexical description.

Part I

White space

- Usually you want to allow white spaces (" ", "\n", "\t") and comments between tokens.
- These are not tokens, and not members of the alphabet of the syntactic description.
- But they are members of the alphabet of the lexical description, and has to be handled by the compiler.

Next part...