(ISBN: 0-471-69466-5)

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Room</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 25 Feb</td>
<td>10.00-12.00</td>
<td>P2446</td>
<td>Compiling</td>
</tr>
<tr>
<td>Ons 27 Feb</td>
<td>10.00-12.00</td>
<td>P1211</td>
<td></td>
</tr>
<tr>
<td>Thu 38 Feb</td>
<td>10.00-12.00</td>
<td>P1211</td>
<td></td>
</tr>
<tr>
<td>Mon 3 Mar</td>
<td>08.00-10.00</td>
<td>P1211</td>
<td>“Recap”</td>
</tr>
<tr>
<td>Thu 2 Mar</td>
<td>15.00-17.00</td>
<td>P1211</td>
<td>Björn Victor</td>
</tr>
<tr>
<td>Thu 2 Mar</td>
<td>10.00-12.00</td>
<td>P1111</td>
<td>Anna Otto</td>
</tr>
<tr>
<td>Wed 2 Apr</td>
<td>08.00-17.00</td>
<td>?</td>
<td>Exam</td>
</tr>
</tbody>
</table>
Operating Systems
- Process Management
- Memory Management
- Storage Management

Compilers
- Compiling process & Lexical analysis
- Parsing
- Semantic analysis & Code generation
Compiling Process

Expected to produce correct object code for all input in the source language for which it was designed, and one or more error messages, for any other (invalid) input.
Compiler vs Translator

Typical: gcc, javac

Non-typical compiler:

- **latex**: document compiler
  
  transforms to DVI printing commands. Input = document (not program)

- **C-to-Silicon compiler**
  
  generates hardware circuits for C programs, output is lower-level than typical compilers

Translators:

- **f2c**: fortran to C (both high-level)
- **latex2html** (both document)
- **dvi2ps**: DVI-to-PostScript (both low-level)
Efficiency

1. efficient compilation
2. minimal compiler size
3. minimal size of object code
4. production of efficient object code
5. ease of portability
6. ease of maintenance
7. Optimal usability
   including good error diagnostics and error recovery
General structure

- High-level source code
- Compiler
- Low-level machine code
Detailed structure

Analysis

Lexical analysis → Syntax analysis → Semantics analysis

Synthesis

Machine independent code generation → Optimization of machine independent code → Storage Allocation → Machine code generation → Optimization of machine code
Lexical analysis

Definition of the Language should be:

- Precise
- Concise
- Machine readable

All strings that belong to the language: its **syntax**.
Meanings of those strings: its **semantics**.

If finite, list them all and that’s it.
If not, how to represent the strings? How to describe tokens?
Regular Expressions

Example (Language)
- $\{x^n | n > 0\}$
- $\{x^n y^n | n > 0\}$: xy, xxxyyy, xxxxxxxxxxxyyyyyyyy
- $\{x^m y^n | m, n > 0\}$
- $\{x^m y^n | m, n \geq 0\}$: xxx, yyyyyyy, $\epsilon$

Can be defined as Regular Expressions
- $x^* y^*, \epsilon$
- $xx^* yy^*$ or $x^+ y^+$
- $x^* | y^*$
- $(x | y)^*$
- $x | y^*$
- $(aab | ab)^* : \epsilon, aababaab, ababab, aabaabaabab$
RE examples

Example (identifier)
- $l(l|d)^*$
- $l$ is a letter
- $d$ is a digit

Example (Fixed-point number)
- $(d^*d.d^*)|(d^*.dd^*)$
- $d$ is a digit
- Note: requires a digit before and after the point.
RE Problem

Problem?
Regular Expression for $\{x^n y^n | n > 0\}$
(same number of xs and ys, and at least one) ?

Need some extra rules...
$S \rightarrow xSy$
$S \rightarrow xy$

Then $S \Rightarrow xSy \Rightarrow xxSyy \Rightarrow xxxyy$ (derivation)
$S \Rightarrow xxxyy$
RE Definition

Defined inductively by:

- **a**: ordinary character stands for itself
- **ε**: the empty string
- **R|S**: either *R* or *S* (alternation), where *R*, *S* are RE
- **RS**: *R* followed by *S* (concatenation), where *R*, *S* are RE
- **R***: Concatenation of RE *R* zero or more times

\[(R^* = \epsilon | RR | RRR | RRRR | \ldots)\]
RE Shorthands

- $R^+$: $RR^*$ (one or more)
- $R?$: $R|\varepsilon$ (optional)
- [abce]: $(a|b|c|e)$
- [a-z]: $(a|b|c|d|e..|y|z)$
- [^ab]: anything but the listed characters
- [^a-z]: one character not from this range
How to break up text

```cpp
if(b == 0)a = b;
if ( b == 0 ) a = b;

defelse x = 0;
else x = 0;
```

- Rule: Longest matching token wins
- if ties in length: priority over tokens

**Lexer - Lexical Analyzer**

Tool to convert text stream to tokens defined by REs + priorities + longest-matching token rule
**Tokens**: Strings of characters representing the lexical units of the program (such as identifiers, numbers, keywords, operators). Unique or not.

**Regular expressions**: concise description of tokens

**Language** denoted by a regular expression $R$: $L(R)$ = set of strings that $R$ represents
How to accept tokens

We use an Finite Automaton

\[ M = (K, \Sigma, \delta, S, F) \]

- \( K \): set of states
- \( \Sigma \): alphabet
- \( \delta \) (transition function)
- \( S \in K \): start state
- \( F \subseteq K \): set of final states

Note: Turing Machine

2 regular expressions as examples

- identifier: \( \text{letter(letter|digit)}^* \)
- real number: \( (\+|\-|\.)\text{digit}^*.\text{digit digit}^* \)
C code for RE - `letter(letter|digit)*`

```c
#include <stdio.h>
#include <ctype.h>

main(){
    char in;
    in = getchar();
    if(isalpha(in)){
        in = getchar();
    } else {
        error();
    }
    while(isalpha(in) || isdigit(in)){
        in=getchar();
    }
}
```
Finite Automaton - $\text{letter} (\text{letter} \mid \text{digit})^*$
C code for RE - letter(letter|digit)*

```c
#include <stdio.h>
#include <ctype.h>

main()
{
    char in;
    int state;
    state = 1;
    in = getchar();
    while(isalpha(in) || isdigit(in)){
        switch(state){
            case 1: if(isalpha(in){state = 2}else{error();} break;
            case 2: state = 2;
                    break;
        }
        in = getchar();
    }
    return (state == 2);
}
```
Finite Automaton - \((+|−|\cdot)\text{digit}^\ast.\text{digit\ digit}^\ast\)
Best Automaton?

Deterministic (DFA) vs Non-deterministic (NFA)

Optimization: Minimize the FA (NFA ⇒ DFA first)

Out of scope for this class.
Automata suitable for automation ⇒ Lex

`letter`  [a-z]
`digit`   [0-9]
`identifier` `{letter}`({letter}|{digit})*

```%
{identifier} {printf(``identifier recognized\n'');}
%
```

$ lex first.lex (appears lex.yy.c)
$ cc -o firstlex lex.yy.c -ll
$ firstlex < cprog
$ firstlex < cprog > identifiers.txt
At Lexical analysis, the lexer

- breaks down the source code in tokens
- accepts them.

In C, 6 types of symbols:

- **Keywords** const, char, if, else, typedef
- **Identifiers** sum, main, printf
- **Constants** 28, 3.1415927, 017 (octal), 0xFF (hexadecimal)
- **String literals** “Tobias”, “Markus”, “eagle”
- **Operators** +, -, ++, >>, /=, &=
- **Punctuation** {}, [], ..., ;

Additional

- deletes comments
- inserts line numbers
Only concerned with symbol recognition

64 const char typedef >> +

Perfectly correct for the lexer.
Up to the syntax analyser (or parser) now