[Note: These are the notes I use for the lectures. To use these as lecture notes, you need to ignore some irrelevant parts. They should work as a summary of the lecture topics, though.]

1 Intro, lecture 2

Course registration: 20 students have not activated themselves. Please do NOW if you intend to continue!

Compiler books for deeper reading: Compilers: principles, techniques, tools (Aho, Sethi, Ullman, Lam). Not req'd for course!

Our book MCIiJ cost ≈500:- in several Swedish and intl bookshops.

2 Repetition

Compilation done in many *passes*. They should pass few, well documented data structures between each other, and execute one after another, i.e. as passes. Typically a "compiler driver" invokes pass after pass.

Early ones are language dependent, late ones are target dependent (explain target!), middle ones are indep of both.

3 Today

Lexical analysis

Grammars

Parsing, different syntax trees

Exactness of parsing: No strings $\notin L$ should be accepted. No trailing garbage allowed.

4 Lexical analysis

This will be brief, see the course book for details, or the Dragon book page 109-189 if you really love lexical analysis.

4.1 Some theory

Regular Language ¡=¿ Regular expressions

Finite automata (DFA, NFA), i.e. automata with a finite, fixed number of states. Generic term FSM for *finite state machine*.

Regular language, over an alphabet A:

1. The empty language is regular L
2. Given $a \in A$, the singleton language a is regular L
3. If X and Y are RLs, $X \cup Y$, $X|Y$, and $X*$ are RLs

Here | denotes concatenation, $*$ the Kleene star (i.e., repetition).

4.2 Lexical analysis and compilers

How does out language understaning work? "jenecomprendspasdesordinateurs"

Relevance: Lexical elements of (most) HLLs, "tokens" form a Regular Language.

Goal: break source "sequence of bytes" into words aka tokens, to be used by parser.
Ex: int, kaka, "error" '{', 4711

Tokens made with "Longest match".
The FSM should read symbols greedily, i.e., "iffy" should be an ident, not **if** and then ident "fy".

Easy to implement in plain programming language.

Even easier (and more readable) in *lexical analyser generators*. E.g. Flex, JLex, JavaCC, SableCC. (Latter two also parser generators.)

Conceptually: Entire state modeled by position in state graph.

In compiler: Actually a lexer is not strictly an FSM, since for some tokens, typically idents and numbers, it needs a buffer.

Limit in powerfulness: Assume exp with m nested left parens to be matched by m right parens. If FSM has $n < m$ states, it cannot repr. And n is fixed, m is unbounded.

5 Grammars and Parse Trees

TODO:

1. Introduce the concept *context-free grammar*
2. Explain the power of grammars, compare to REs
3. Explain (1) terminal symbols (2) non-terminal variables (3) productions (4) start symbol
4. Explain properties of grammars, i.e. grammar classes (next lesson)
5. Concrete syntax tree (aka Parse trees) vs Abstract syntax trees

Cf REs and context-free grammars:

Reason well about the term 'language' L in its CS abstract sense. Every RE define a *language*, every [context-free] grammar define a *language*.

Regexps have concatenation, repetition, alternation. But they permit no recursion in their defs.

Grammars add recursion; this adds immensive expressiveness.

5.1 Example 1

Let's match parentheses.

$$S \to SS$$
$$S \to (S)$$
$$S \to ()$$

Let S be start symbol (among the grand choice).

How do we go about to match a string to the grammar?

(1) Choose any *production*
(2) Try to match your string to the rhs
(3) If it matches–recursively–then we're done, else try another rule

Complexity of this algorithm with n productions. . . ?

Match these strings of the *language* to the grammar:
"()" "()()" "((()(()))))".

Does the empty string belong to this language? Why not?

Repeat why FSM cannot handle this language.

5.2 General context-free grammar formalism

We have

1. A set T of *terminal symbols* (typically from our lexer)
2. Set of *non-terminal* symbols/variables V, each member defining a (sub-) language
3. *Productions*, i.e., exps like $nontermsymbol \rightarrow symbolsymbol \cdots symbol$
4. A start variable (i.e., some of the non-terminal)

Carefully explain these concepts.

5.3 Example 2

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow integer$$

Let E be start symbol.

What now, we have "left recursion"...! (Next lecture.)

Show syntax tree(s).

5.4 Next lecture:

Different derivations, leftmost derivation, rightmost derivation, ambiguous grammars, syntax trees

6 Parse Trees and Grammars

Gen property: A subtree is connected to the node from which is was derived.

Show example of 12 + 34 + 56 using the last grammar.
Show example of 12 - 34 - 56 using the last grammar.

A grammar is ambiguous if > 1 different parse trees can be derived for at least one string in the language. Our grammar is indeed ambiguous (and this is a problem since it does not encode the lack of associativity of the $-$ operator).

Problem since we intend to assign meaning from parse trees.

Now let's study a more complete grammar for arithmetic expressions on integers. The terminal symbols are *integer* and $+$, $-$, $*$ and $/$.

$$
\begin{array}{lcl}
E & \rightarrow & \mathit{integer} \\
E & \rightarrow & E + E \\
E & \rightarrow & E - E \\
E & \rightarrow & E * E \\
E & \rightarrow & E \ / \ E \\
E & \rightarrow & (\ E\ )
\end{array}
$$

Claim: This recognises the *language* we expect for integer expressions.

But: It is highly ambiguous. Neither associativity nor precedence is encoded.

Alternative grammar, introducing the non-terminals T and F:

$$
\begin{array}{lll}
E \rightarrow T & T \rightarrow F & F \rightarrow \textbf{integer} \\
E \rightarrow E + T & T \rightarrow T * F & F \rightarrow (E) \\
E \rightarrow E - T & T \rightarrow T/F &
\end{array}
$$

Discuss why this is better: rhs'es using a "lower" symbol cannot expand to same syntactical elements as the current production. I.e., T cannot expand to anything with + (except within parens!).

7 End

Distribute and introduce Minijava grammar. Briefly go through it. [not done]

Next week:

1. Next lecture: D3.

2. Parser generators
3. Grammar classes and LL, LR parsing
4. Semantic actions