Norwegian University of Science and Technology

Compiler Construction

Lecture 25: Semester review

Michael Engel

First things first – what's relevant?

- **The most urgent question: which parts of the material in the course is relevant for the exam?**

- **Everything except:**
  - Details of attribute grammars in lecture 11
  - Details of the ELF file structure in lecture 15
  - Lecture 24 (Static Single Assignment)
- **Especially these topics are relevant:**
  - Topics of the practical exercises (!)
    - You should know your yacc and lex…
  - The material from the lectures as rehearsed in the theoretical exercises

What about…?

- The C crash course?
  - The exam will certainly require knowledge of C on an advanced level, but we won't ask specific questions about details of C programming
  - Nevertheless, you must be able to read and understand C programs and also to find and fix errors
- VSL?
  - You don't need to learn the VSL syntax. Whenever a question is related to VSL (or any other language constructs), you will be given the relevant details along with the question

The rest of this lecture

This is a quick walk-through summary of the ***most relevant*** topics we discussed in the run of this course

It gives an overview of the **minimum** extent of (sub)areas I expect you to have knowledge about

…but it is certainly not exhaustive

The takeaways from each lecture are given as a number of ***questions*** – you can figure out if you can answer these and, especially if not, read up on these topics

As mentioned before, lecture 24 is not relevant for the exam and is thus not summarized here

1. Motivation and History

Introduction to compilers and some history of programming languages. Most of this should serve as motivation.

Important questions:

- What is the difference between machine language and assembler?
- What is missing in assembler compared to high-level languages?
- Which components are part of the compilation process?
- What is a transpiler?

You do *not* need to know details about the historical compilers and languages discussed in that lecture

2. Compiler Structure and Lexical Analysis

Let's dig into the topic!

- What is the structure (stages) of a typical compiler?
  - What does each stage do and what are the in/outputs?
- What is an intermediate representation (IR) and why is it useful?
- What is the task of lexical analysis?
  - What is a lexeme, what is a token?
  - Which classes of lexemes exist?
  - How is "semantic" defined?
- What is the structure of a (deterministic) finite automaton (DFA)?
  - Can you transform between graph and table representations?
  - Can you implement a DFA in C with and without tables?
  - What is a scanner generator and why is it useful?

3. Scanner Generators

- What is an alphabet and a language related to a DFA?
- What does "accepting a language" mean?
- Which operations on languages exist?
- What are regular expressions and how are they written?
- What is the relation of regular expressions and DFAs?
  - Can you transform one into the other?
- What is a regular language?
- How can you combine simple automata to build a DFA to accept a regular language?
- What is a non-deterministic finite automaton (NFA)?
  - Can you explain the difference to a DFA?
- How can you construct a scanner using an NFA?
  - What is a Kleene closure?

4. Lexical analysis in the real world

- How can you convert an NFA into a DFA?
  - Subset construction algorithm
- How can you minimize the states of a DFA?
  - Hopcroft and Myhill-Nerode algorithms
- How does lex work?
  - What is the input syntax?
  - How do you use states and hierarchy?
- What is a greedy automaton?

5. Introduction to Parsing

- What is parsing?
- How do scanner and parser interact?
- What is a context-free language?
- What are top-down and bottom-up parsing?
- Why are regular expressions not powerful enough to parse context-free languages?
- How can we specify a context-free grammar?
- How can we derive a language from its grammar?
- Why can grammars be ambiguous and what can you do about it?
- What is the Chomsky hierarchy?

6. Top-down parsing and LL(1) parser construction

- What is lookahead related to parsers?
- What are terminal and nonterminal symbols in a grammar?
- What is left and right recursion and left/right factoring?
- How can you eliminate left recursion?
- How does recursive descent parsing work?
  - What is backtracking and how does it work?
- How can you implement a recursive descent parser in C?
  - How do you handle recursive grammar constructs?
- How does table-driven parsing work?
  - What are the FIRST and FOLLOW sets?
  - What is nullability?
  - What is a LL(1) parser and how do you construct its table?

7. Bottom-up parsing

- What does the (parse) stack look like for recursive descent parsing?
- Can you explain the difference between top-down and bottom-up parsing?
  - Which parts are already known, which actions can be taken when, which lookahead is important?
- What is the general idea of bottom-up parsing?
  - What do the parse trees look like?
- What are shift and reduce actions in a bottom-up parser?
- How is the parse stack used in bottom-up parsers?
  - What stack operations are performed by shift and reduce?

8. LR-parsing

- How can we make the correct decisions in bottom-up parsers?
- How does LR parsing extend the idea of bottom-up parsing?
- What are LR(1) items?
- How can you construct a DFA for an LR parser?
  - How do you construct the LR table?
  - What is the relation of the DFA and the LR table?
  - Can you explain the LR parsing algorithm?
- How do you use the LR table for parsing?
  - What is a conflict in an LR table?
  - What are shift-reduce and a reduce-reduce conflicts?
  - How can you use precedence rules to solve conflicts?
- Which variants of LR parsers exist?

9. Practical parsing issues and yacc intro

- How does error recovery work?
- How can you handle unary operators such as "-"?
- What is context-sensitive ambiguity and how can you handle it?
- What is the impact of left vs. right recursion in a grammar?
- What is the syntax of a yacc specification?
- How do lex and yacc interact?
- How do you define tokens, precedences, rules in yacc?
- What do the yylex() and yyparse() functions do?
- What does a yacc grammar action look like?
  - What do the "variables" $$, $1, $2, … stand for?

10. Context-sensitive analysis

- What is semantic analysis?
  - Why can semantic not be easily described in a grammar?
- What is the task of context-sensitive analysis?
  - Can you give examples for non-syntactical information?
- What is a symbol table, what are its contents?
  - How can you implement symbol tables?
  - Why are hash tables useful to implement symbol tables?

11. Type systems and attribute grammars

- What are type systems and what are they used for?
- What is type safety?
  - Which advantages and drawbacks does it have?
  - How does a compiler perform type checking?
- What is the relation of type checking and expressiveness?
- What are the components of a type system?
  - What are compound and constructed types?
  - How can a compiler determine type equivalence?
- How can a type system help to create better code?
- What is an attribute grammar?
  - *We won't go into details of attribute grammars in the exam!*

12. Intermediate representations and three-address code

- Which sorts of intermediate representations exist?
- What is a syntax tree, what is an AST?
  - How can you derive an AST from a syntax tree?
- What is a DAG?
  - How are DAGs related to ASTs, what is their advantage?
- What is a control-flow graph (CFG)?
  - Where in a compiler are CFGs used?
- What is a dependence graph?
  - What is use and def of a value/variable?
- Can you construct CFGs and dependence graphs?
- What is a linear IR, can you give examples?
- What is three-address code (TAC)?

13. Intermediate representations and SSA

- How can you represent linear IRs (especially TAC) in a compiler?
  - What are the benefits/drawbacks of the different approaches?
- What is the format of TAC?
  - Which instructions/operations exist in TAC?
  - How is control flow represented?
  - How can you translate from an AST to TAC? (linearizing)
    - Expressions, control structures, loops…

14. The procedure abstraction

- What are the components of a procedure/function?
  - What are the benefits of using them?
- What is a name space and a scope of a variable?
- How do procedures integrate into the control flow?
- How are procedures implemented?
  - Which code has to be generated at compile time?
  - What is done at runtime?
- What is an activation record, what is its structure?
  - How does recursive invocation of procedures work?

15. x86-64 and real world procedures

- Which registers to x86-64 CPUs support?
- Which addressing modes exist?
- How is control flow (if/while) implemented in x86-64 asm?
- How are procedure calls implemented?
  - How do we pass parameters and return values?
- How do we handle activation records and the procedure context?
  - Which state has to be preserved (saved) by the caller/ callee and where is it stored?
  - How are local variables handled and how does the x86-64 stack work?

16. Introduction to optimizations

- How is "optimization" defined in a compiler context?
  - What is the difference to mathematical optimization?
- Why do we need/want optimizations?
  - Which optimization objectives exist?
- Why are optimizations commonly applied on IR level?
- Can you explain and perform simple optimizations?
  - Constant folding
  - Algebraic simplification
  - Strength reduction

17. Optimizations in detail

- How do we define a safe optimization?
- How do we detect dead code?
  - Why is this more complicated when control structures and/or loops are involved?
  - Can you give examples for dead code?
- What is a control-flow graph (CFG), how to construct one?
  - How are basic block defined, can you split a program into basic blocks (BBs)?
  - What are infeasible executions in a CFG?
- How does liveness analysis work using CFGs?
  - What is a program point?
  - What are in/out and def/use sets?
  - Can you perform liveness analysis on a given CFG?

18. Data flow analysis framework

- Can we be sure to achieve convergence in our analyses?
- What is the precision of a data flow analysis?
- What are sets and (partial) orders?
  - What is a Hasse diagram, how can you use it to describe partial orders?
- Can you define the least upper bound (LUB) and greatest lower bound (GLB)?
- What is a lattice, what are meet and join operations?
- What are power sets, what is a power set lattice?
- How can we use lattices to perform live variable analysis?

19. Data flow analyses and Live variable analysis

- Optimizing analyses: how do bit vectors represent sets?
  - How do set operations translate to binary operations?
  - How can we use bit vectors to represent graphs?
- How can we discover data flow information?
- What are data flow equations, how can we construct them?
- Why do we need two directions in data flow analysis?
  - How do Gen and Kill sets relate to bit vector analysis?
- How can we use our data flow analysis framework to perform live variable analysis (a backward analysis)?
  - How are Gen/Kill and In/Out sets defined here?
  - How can we compute use and def for a basic block?
- What are applications of liveness analysis?

20. Reaching definitions

- What is a reaching definitions analysis?
  - What do the related data flow equations look like?
  - What are def-use and use-def chains?
- Which applications do reaching definitions analyses have?

21. Available exp. and very busy expressions

- How can we discover global properties of expressions?
- What is availability (of an expression)?
  - What is the difference between semantic and syntactic analysis here?
- How does available expressions analysis work?
  - What do the related data flow equations look like?
  - What can it be used for?
- What is a very busy expression analysis?
  - What do the related data flow equations look like?
  - What can it be used for?
- What are must- and may-analyses, can you give examples?

22. Code generation

We're almost done with the compiler… on to the backend!

- What is instruction selection?
- How can we translate IR code to machine code?
  - Why can there be alternative translations?
- How does tiling of syntax trees work?
  - How is instruction selection related to tiling?
- How can DAGs be used for optimization in code generation?
  - Why is a DAG more efficient than a syntax tree here?

23. Register allocation

The last bit of the backend…

- Why is it useful to keep variables in registers?
  - What is the major problem related to this?
  - Which data cannot be kept in a register?
- How can we decide which variables to keep in registers?
  - How can we determine conflicts when two variables share one register?
- How is live variable analysis used to construct an interference graph?
- What is graph coloring and what it its complexity?
- What is spilling and when is it used?
- What are precolored nodes in the interference graph used for?

That's all…

If you are *still* interested in compiler topics 😉, get in touch for…

- Projects
- Master theses
- Work as student research assistant
- …or just to chat about compiler/language/machine-level programming… topics, of course!