Linköpings universitet
IDA Department of Computer and Information Sciences
Prof. Peter Fritzson and Doc. Christoph Kessler

TENTAMEN / *EXAM*

TDDB29 Kompilatorer och interpretatorer / *Compilers and interpreters*

TDDB44 Kompilatorkonstruktion / *Compiler construction*

17 dec 2005, 08:00–12:00

Jour: Christoph Kessler (070-3666687, 013-282406)

Hjälpmedel / *Admitted material:*

– Engelsk ordbok / *Dictionary from English to your native language*;
– Miniräknare / *Pocket calculator*

General instructions

- This exam has 10 assignments and 7 pages, including this one.
  Read all assignments carefully and completely before you begin.
- The first assignment (on formal languages and automata theory) is ONLY for TDDB29, while the last one (on code generation for RISC...) is ONLY for TDDB44.
- It is recommended that you use a new sheet for each assignment. Number all your sheets, and mark each sheet on top with your name, personnummer, and the course code.
- You may answer in either English or Swedish.
- Write clearly. Unreadable text will be ignored.
- Be precise in your statements. Unprecise formulations may lead to a reduction of points.
- Motivate clearly all statements and reasoning.
- Explain calculations and solution procedures.
- The assignments are *not* ordered according to difficulty.
- The exam is designed for 40 points (per course). You may thus plan about 5 minutes per point.
- Grading: U, 3, 4, 5. The preliminary threshold for grade 3 is 20 points.
- **OBS C:are antagna före 2001:** Om du vill ha ditt betyg i det gamla betygsystemet (U, G, VG) skriv detta tydligt på omslaget av tentan. Annars kommer vi att använda det nya systemet (U, 3, 4, 5).

Solution proposal:
We give solution proposals for the assignments below. Note that, in some cases, there may be more than one possible correct answer; we show for brevity only one or a few of these. ◇

1. **Only TDDB29:** (6 p.) **Formal languages and automata theory**

   Consider the language L consisting of strings w over the alphabet $\{a, b\}$ such that w contains at least 3 a's (not necessarily in sequence) and not more than one b.

   (a) Construct a regular expression for L. (1p)

   Solution proposal:
   $e = aaa^+|aa^+ba^+|a^+baa^+|aaa^+b|baaa+$ ◇

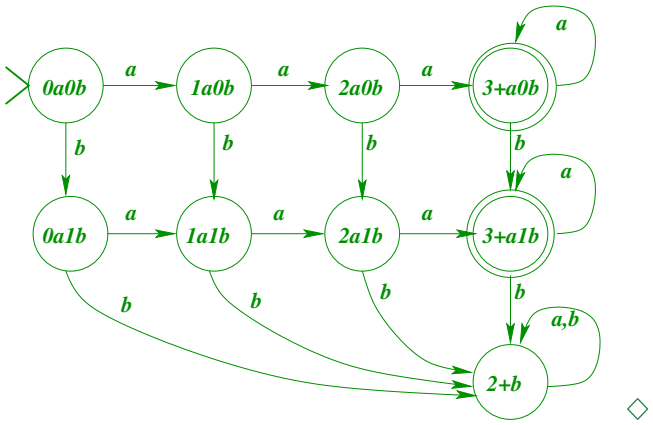
   (b) Construct from the regular expression an NFA recognizing L. (1p)

   Solution proposal:
   Follow the recursive construction rules as presented in the lecture. ... ◇

   (c) Construct a DFA recognizing L, either by deriving from the NFA of question (1b), or by constructing one directly. (3p)

   Solution proposal:
   One possibility uses the construction from the NFA, via ϵ-closures and subset construction. There is also a direct DFA construction method from regular expressions described in the dragon book, which was not covered in the lecture. The third one is to build a DFA from scratch, without considering the regular expression at all. As an illustration, we sketch the DFA derived by the third variant:

   The DFA must always know how many a's and b's have been encountered. Hence, we need the following states: $\langle 0a0b\rangle$, $\langle 1a0b\rangle$, $\langle 2a0b\rangle$, $\langle \underline{3^+a0b}\rangle$, $\langle 0a1b\rangle$, $\langle 1a1b\rangle$, $\langle 2a1b\rangle$, $\langle \underline{3^+a1b}\rangle$, $\langle 2^+b\rangle$. The underlined states are accepting states, $\langle 0a0b\rangle$ is the start state, and $\langle 2^+b\rangle$ is an error state. The transitions are as follows:

   ◇

   (d) We have learned that "finite automata cannot count". Give an example of a formal language that cannot be recognized by a finite automaton. (0.5p)

   Yet, for the language L above there exists a finite automaton recognizing it. Why is this not a counterexample? (0.5p)

   Solution proposal:
   The language $\{a^n b^n,\ n \in \mathbb{N}\}$ cannot be described by a finite automaton. The example above is not a counterexample because it only counts, via different states, up to a finite value. ◇

2. (4p) **Phases and passes**

   (a) What is a phase? (0.5p)

   Solution proposal:
   A phase is a step in the compiler chain solving a specific task, such as lexical analysis or code generation. A phase gets (part of) its input directly from the previous phase. For instance, the parser gets its input by calling the lexical analyzer to provide the next token. ◇

   (b) What is a pass? (0.5p)

   Solution proposal:
   A pass is a complete sweep of the input program from start to end. Simple compilers such as `lcc` can do everything within a single pass, while for some advanced compilation techniques such as interprocedural analysis it is necessary to do multiple passes. For instance, a first pass may collect global analysis information that is then used by subsequent passes. ◇

   (c) Describe what phases normally are found in a compiler, what is their purpose, how they are connected, and what is their input and output. (3p)

   Solution proposal:
   See the slides in Lecture 1 and Section 1.3 of the course book. ◇

3. (5p) **Top-Down Parsing**

   (a) Given a grammar with nonterminals S and X and the following productions:
   $S \rightarrow aSc \mid aX$
   $X \rightarrow Xb \mid d$
   where S is the start symbol.

   What is/are the problem(s) with this grammar if it is to be used for writing a recursive descent parser with a single token lookahead? Resolve the problem(s), and write a recursive descent parser for the modified grammar. *(Pseudocode is fine. Use function* `scan()` *to read the next input token.)* (4.5p)

   Solution proposal:
   Left factoring is needed in the first rule to peel off the leading a.
   $S \rightarrow aT$
   $T \rightarrow Sc \mid X$
   This is necessary for being able to select the right rule with a single lookahead token.
   The second rule has left recursion, which can easily be modified to
   $X \rightarrow dY$
   $Y \rightarrow bY \mid \epsilon$
   From this, construction of a recursive descent parser is straightforward, following the recipe of the lecture. ◇

   (b) The theory for formal languages and automata says that a stack is required for being able to parse context-free languages. We have used such a stack, for instance, in the

LL-item pushdown automaton in the lecture on top-down parsing. But where is the corresponding stack in a recursive descent parser? (0.5p)

Solution proposal:
in the compiler's runtime stack, which holds the activation records for the called functions of the recursive descent parser, each containing information on the nonterminal and where it occurred (PC return address) on a right-hand side of a grammar rule. Note that the stack holds the entire chain of nonterminals expanded on the path to the current position (following backwards the chain of pointers to caller's activation record, starting from the current one). ◇

4. (7 p.) **SLR(1) parsing**

Given the following grammar:

```
1. <S> ::= a <S> a
2.       |  <S> b
3.       |  b
```

(a) Construct the canonical LR(0)-items and the LR(0)-item DFA for the grammar. (3p)

(b) Use the states and state transitions of (a) for constructing action and goto tables. (2p)

(c) Show how the string `abba` is parsed. (2p)

5. (3 p.) **Memory management**

What is an activation record? What properties of a programming language lead to a need for activation records? What does an activation record contain?

Solution proposal:
An activation record is a data structure, typically allocated as a contiguous block of memory locations on the stack, that contains all information required for executing a procedure call:

Call arguments, return address of the caller, link to the caller's activation record (i.e., dynamic link) and, where applicable, to the statically enclosing procedure's activation record; saved registers of the caller, local variables of the callee, and (if the procedure returns a value) a memory location where the return value can be stored. [As an optimization to save memory accesses, some of these values (e.g., a small number of arguments and the return value) may, under certain conditions, be passed in special registers instead.]

Languages that do not support recursion may statically pre-allocate one activation record for every procedure in a predetermined memory area, as there can only be at most one activation of a procedure at any time. Languages that do not have procedures do not need activation records at all. ◇

6. (7 p.) **Intermediate code generation**

(a) Translate the following code segment into quadruples, postfix code, and abstract syntax tree: (4.5p)

```
x = 123;
y = 3;
while (x>100) {
  x = x - y;
  y = 2*y;
}
```

(b) Given the following program fragment in quadruple form:

```
1:  T1 := a + b
2:  T2 := T1 - c
3:  x  := T2
4:  T3 := x > 0
5:  if T3 goto 10
6:  T4 := x + 1
7:  x  := T4;
8:  T5 := a + b
9:  goto 4
10: m := T5
```

Divide this program fragment into *basic blocks* and then draw the *control flow graph* for the program fragment. (2.5p)

7. (3 p.) **Code generation for trees**

In the lecture on code generation, we considered the special case of DAGs that are trees (that is, the result of each quadruple operation is used only once). With the *labeling* algorithm [Sethi/Ullman 1970], it is possible to generate *space-optimal code* for trees, i.e., a code sequence that uses a *minimum* number of registers. For the following quadruple sequence, show the tree form, apply the labeling algorithm to generate space-optimal code (given in quadruple form again), and assign registers (`r1,r2,` ...) to the temporaries (`T1,T2,`...).

```
1:  T1 := a + b
2:  T2 := c * d
3:  T3 := e - f
4:  T4 := T2 * T3
5:  T5 := T1 / T4
```

Solution proposal:
See the lecture slides and course book for the Sethi/Ullman labeling algorithm. Draw the tree and annotate the nodes with the labels according to the label formula of the lecture. The root gets label 2, which means that a space-optimal evaluation (one that, recursively, always evaluates the subtree with larger label value first) needs exactly 2 registers. Construct such an evaluation (there may be several possibilities when subtrees of a node have equal label values), show the live ranges of the temporaries T1, T2, ..., and assign registers r1, r2 to the temporaries in a greedy way. ◇

8. (2 p.) **Loop optimizations**

   Describe the loop transformation *loop unrolling*.

   What are the potential benefits and drawbacks of its application? (2p)

   Solution proposal:
   Description: see lecture notes.

   Advantage: Reduction of loop overhead (branches – expensive!).

   Disadvantage: Code expansion, may be a problem especially in embedded systems. ◇

9. (3 p.) **Bootstrapping**

   Explain the concepts of *rehosting* and *retargeting*. Use T-diagrams.

10. **Only TDDB44:** (6 p.) **Code generation for RISC ...**

    (a) Explain the main similarity and the main difference between superscalar and VLIW architectures from a compiler's point of view. (1.5p)

    Solution proposal:
    See lecture notes for lecture 10, Comparison... Both are instruction level parallel architectures with multiple functional units, which puts high demands on instruction scheduling to perform efficiently. While a superscalar processor works on an ordinary, linear stream of instructions, a VLIW machine processes long instruction words that contains multiple, explicitly parallel instructions in packed form. VLIW is harder to generate code for, because the dependences must be analyzed and code scheduled statically. These tasks are (at a certain degree) done automatically at run time in the superscalar processor's dispatcher. ◇

    (b) Given the following medium-level intermediate representation of a program fragment (derived from a `for` loop):

```
1:   c = 3;
2:   k = 20;
3:   if k<=0 goto 9;
4:   a = c / 2;
5:   b = a + c;
6:   c = a * b;
7:   k = k - 1;
8:   goto 3;
9:   d = b * c
```

    Identify the live ranges of program variables, and draw the live range interference graph

    (i) for the basic block in lines 4–7 (i.e., the loop body),

    (ii) for the entire fragment.

    For both (i) and (ii), assign registers to all live ranges by coloring the live range interference graph. How many registers do you need at least, and why? (3.5p)

    Solution proposal:
    Very similar to the example in Lecture 10. ◇

(c) Register allocation and instruction scheduling are often performed separately (in different phases). Explain the advantages and problems of this separation. (1p)

Solution proposal:

See lecture 10, "Conflict with instruction scheduling".

From a compiler (software) engineering point of view, it is easier to handle these tasks in separate phases, such that each can be developed and tested separately. However, this separation leads to "artificial" constraints on the code; the phase run first constrains the other one. For instance, doing register allocation first may introduce "false" data dependences that constrain the instruction scheduler. Vice versa, scheduling instructions first may lead to excessive register need, which in turn may require spilling, and this spill code must be scheduled again... ◇

Good luck!