Sample Exam
Compiler Construction
Summer 2023

Dr. Arnd Hartmanns

Main exam: 22 June / Resit: 7 July 2023

Example Solution and

Name

Grading Scheme

Student number Seat

Do not open this exam booklet before we ask you to. **Do** read this page carefully.

You can only write the exam at the seat allocated for you, and you must use the exam booklet that carries your name and student number.

This is a closed-book exam. Leave bags and jackets out of reach. Turn off all electronic devices and leave them with your bag. You may only take writing utensils, drinks, food, and your student or identity card to your seat. Please have your student or identity card clearly visible on your table.

Leaving your table or the room with your exam booklet is regarded as an attempt of deception. You may not leave the room during the first 30 minutes and the last 15 minutes of the exam. If you need to use the restrooms, please alert the supervisor. Only one person at a time may leave for the restrooms.

Before you start, please check that your exam booklet consists of **13 pages**, sequentially numbered, on 13 pages sheets of paper, and contains questions 1 through 7 b). This is page 1.

Write your solutions on the (printed) right pages of the exam booklet, in the space provided below the respective questions. Solutions written in a language other than English, in red or similar colours, on the first or last page of the booklet, on the (blank) left pages, or on additional sheets not referenced from the booklet, **will not be graded**. Should you run out of space, ask the supervisor for an additional sheet of paper. You may use a pencil.

The duration of the exam is **120 minutes**. The total number of points of the questions in this exam is 120. To pass the exam, 60 points will be sufficient.

Have fun!

1	2	3	4	5	6	7		Sum
6	15	25	9	16	38	11		120

Question 1. True or False? (6 points)

For each of the following statements, decide whether it is true or false. If a statement is false, very briefly (i.e. in at most one sentence) correct it or explain why it is false.

a) Deciding whether a context-free grammar is ambiguous is an NP-hard problem.

1 False:
1 It is undecidable.

b) 2-address code linear IRs are no longer in common use today.

2 True.

c) When using a global display for addressability, storing the display in global memory works well for multi-threaded programs.

1 False:
1 For multithreaded programs, the display needs to be in thread-local memory.

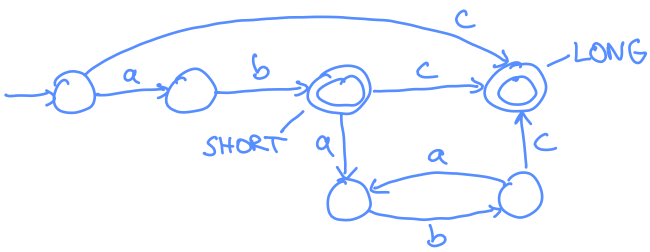
Question 2. Scanning (15 points)

Consider a language over the alphabet { a, b, c } with two token types defined by regular expressions as follows:

1. Token type `SHORT` with regular expression `ab`
2. Token type `LONG` with regular expression $(\texttt{ab})^*\texttt{c}$

a) Draw one DFA that recognises tokens of both types. Mark each accepting state with a double outline ◎ and annotate it with the corresponding token type. *(8 points)*

(You do not need to draw edges for invalid inputs: we assume that input characters that do not have an edge lead to a non-accepting invalid token *state.)*



1 point each for:
- accepting ab
- accepting c
- accepting abc
- accepting $(ab)^+c$
- annotation SHORT
- annotation LONG

+2 P. for not accepting anything else

Now assume we have embedded an implementation of such a DFA into a scanner that repeatedly calls function `NextWord` (as in the lectures and the book, repeated below in three columns) until it reaches the end of the input or finds an invalid token.

```
NextWord()
  state ← s0 ;
  lexeme ← " ";
  clear stack;
  push(bad);

  while (state≠se) do
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ SA
       then clear stack;
    push(state);

    cat ← CharCat[char];
    state ← δ[state,cat];
  end;

  while(state ∉ SA and
        state≠bad) do
    state ← pop();
    truncate lexeme;
    RollBack();
  end;

  if state ∈ SA
    then return Type[state];
    else return invalid;
```

b) For each of the following inputs, state whether it is fully scanned successfully. If yes, what are the resulting tokens and their token types? *(7 points)*

(i) `ababcab` Yes. (1) Tokens: ababc, ab Types: LONG, (1) SHORT (1)

(ii) `cbabab` No. (1)

(iii) `ababab` Yes. (1) Tokens: ab, ab, ab Types: SHORT, SHORT, SHORT (2)

Question 3. Parsing (25 points)

The following grammar, in ANTLR-like notation, is for expressions with binary addition (+) and subtraction (-) operators over single-letter identifiers of token type `NAME`. The start symbol is `expr`.

```
1 expr: '(' expr ')'
2     | expr op NAME;
3     | NAME;
4 op: '+' | '-';
```

a) What is the associativity of the two operators according to this grammar? Mark the correct answer. *(1 point)*

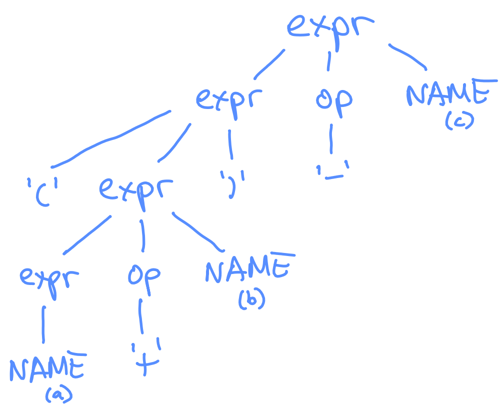
☐ + and - are right-associative.

1 ☒ + and - are left-associative.

b) Is the grammar LL(1)? Briefly but precisely say why or why not. *(3 points)*

1 No,
1 because of the left-recursion
1 on expr in rule 2.

c) Draw the parse tree for the expression "`(a + b) - c`". *(6 points)*



Starting from the root,
+1 for every correct
rule application.
Do not follow incorrect
rule applications.

Now consider the following grammar in BNF, where uppercase letters are nonterminals and lowercase latin letters are terminals, and A is the start symbol:

$$\begin{aligned} A &\rightarrow B\ C \\ &\mid a \\ B &\rightarrow C\ b \\ &\mid \epsilon \\ C &\rightarrow c \\ &\mid \epsilon \end{aligned}$$

d) Give the FIRST and FOLLOW sets for all nonterminals, and the FIRST+ sets for all rules. *(12 points)*

FIRST(A) = { a, b, c, ε }

FIRST(B) = { b, c, ε }

FIRST(C) = { c, ε }

FOLLOW(A) = { eof }

FOLLOW(B) = { eof, c }

FOLLOW(C) = { eof, b }

Boolean grading:
+1 for every correct set,
+0 for every incorrect set no matter what the mistake is.

FIRST+($A \rightarrow B\ C$) = { eof, b, c }

FIRST+($A \rightarrow a$) = { a }

FIRST+($B \rightarrow C\ b$) = { b, c }

FIRST+($B \rightarrow \epsilon$) = { eof, c }

FIRST+($C \rightarrow c$) = { c }

FIRST+($C \rightarrow \epsilon$) = { eof, b }

Boolean grading like above, but what is correct is determined by computing FIRST+ from the FIRST & FOLLOW sets given above (i.e. do not penalise inherited errors).

Don't care about ε being left out here (as in the reference solution), but incorrect εs lead to 0 points for set.

e) Is the grammar LL(1)? Briefly but precisely say why or why not. *(3 points)*

1 No,
1 because the FIRST+ sets overlap (both containing c)
1 for the two rules of nonterminal B.

Question 4. Elaboration (9 points)

The following grammar in ANTLR-like notation with start symbol `number` defines a language for non-negative binary numbers:

```
number: '0b' list;
list: list bit
    | bit;
bit: '0'
   | '1';
```

We want to compute the decimal values of such numbers; for example, the decimal value of `0b0101` is 5.

a) We use one attribute *decVal* for all nonterminals. Below, for each grammar rule, give an attribute rule so that `number`.*decVal* is computed as the decimal value of the number. *(6 points)*

Grammar rule	*Attribute rule*	
`number: '0b' list`	number.decVal $\leftarrow$ list.decVal	1
$\texttt{list}_1$`:` $\texttt{list}_2$ `bit`	list_1.decVal $\leftarrow 2 \cdot \text{list}_2$.decVal + bit.decVal	2
`| bit`	list_1.decVal $\leftarrow$ bit.decVal	1
`bit: '0'`	bit.decVal $\leftarrow 0$	1
`| '1'`	bit.decVal $\leftarrow 1$	1

b) Are your rules synthesised or inherited? If you have both types of rules, say which of them are synthesised and which are inherited. Briefly justify your answer. *(3 points)*

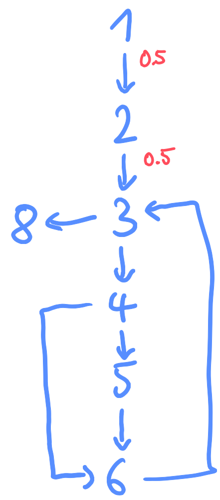
1 All rules are synthesised
1 because they all compute the value of a (parent) node
1 based (only) on the values of its child nodes.

Question 5. Graph-Based IRs (16 points)

Consider the following Java-like code snippet, which finds the maximum element ≥ 0 contained in array `a`:

```
1  int max = 0;
2  int i = 0;
3  while(i < a.length) {
4    if(a[i] > max)
5      max = a[i];
6    i = i + 1;
7  }                     max
8  print("Max:␣", result);
```

a) Draw the control flow graph of the code snippet. Use the line numbers as the nodes of your graph, ignoring line 7. *(7 points)*



+1 for every other correct edge,
−1 for every incorrect edge.

Ignore edges without arrow tips.

b) List all basic blocks of the snippet's control flow graph that consist of more than one node. *(2 points)*

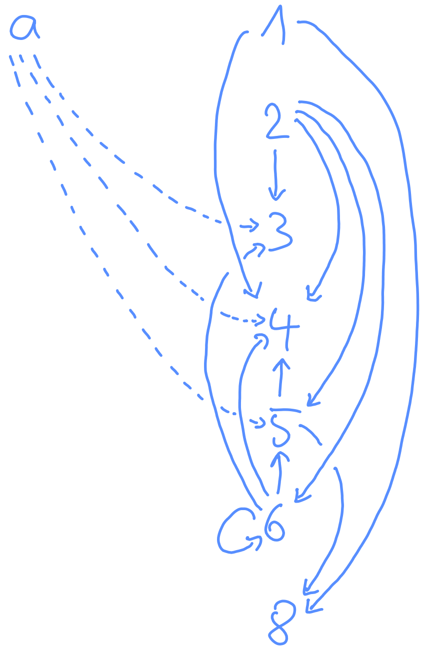
{1, 2} only

2
Boolean grading
(=0 if wrong basic block or more than 1 basic block)

We recall the code snippet from the previous page:

```
1 int max = 0;
2 int i = 0;
3 while(i < a.length) {
4   if(a[i] > max)
5     max = a[i];
6   i = i + 1;
7 }
8 print("Max:␣", max);
```

c) Draw the data dependence graph for the code snippet. Again use the relevant line numbers as nodes, and add a node for the array a. *(7 points)*



+0.5 for every correct edge (with arrow tip),

−0.5 for every incorrect edge.

Note: Don't rely on "points = 2× edges" as a hint in the exam: (a) we make mistakes, too, and (b) the grading scheme may be different!

Question 6. Procedures (38 points)

Consider the Pascal program below on the left, in which only procedure calls and variable declarations are shown:

```
program Main;
  var x, y, z: integer;
  procedure A;
    var x: integer;
    begin { body A } end;
  procedure B;
    var y: real;
    procedure C;
      var z: real;
      procedure D;
        var y: real;
        begin {body D }
          A;
        end;
      begin { body C }
        A; D;
      end;
    begin { body B }
      C;
    end;
  begin { Main }
    B;
  end.
```

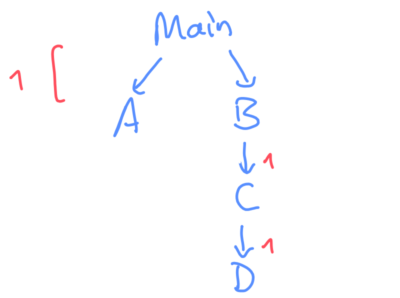
a) Fill the following static coordinate table for the program. *(8 points)*

scope	x	y	z
main	(1, 0)	(1, 4)	(1, 8)
A	(2, 0)	(1, 4)	(1, 8)
B	(1, 0)	(2, 0)	(1, 8)
C	(1, 0)	(2, 0)	(3, 0)
D	(1, 0)	(4, 0)	(3, 0)

Point given if scope level correct (can be 0- or 1-based, solution is 1-based) and offsets chosen so that variables don't overlap.

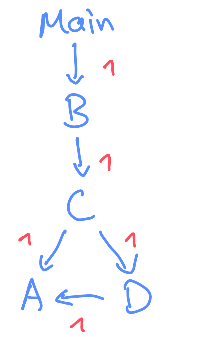
b) Draw the program's scope and call graphs. *(8 points)*

Scope graph:



In the scope graph, arrow tips can be omitted if direction is clear (e.g. laid out as a tree as above).

Call graph:



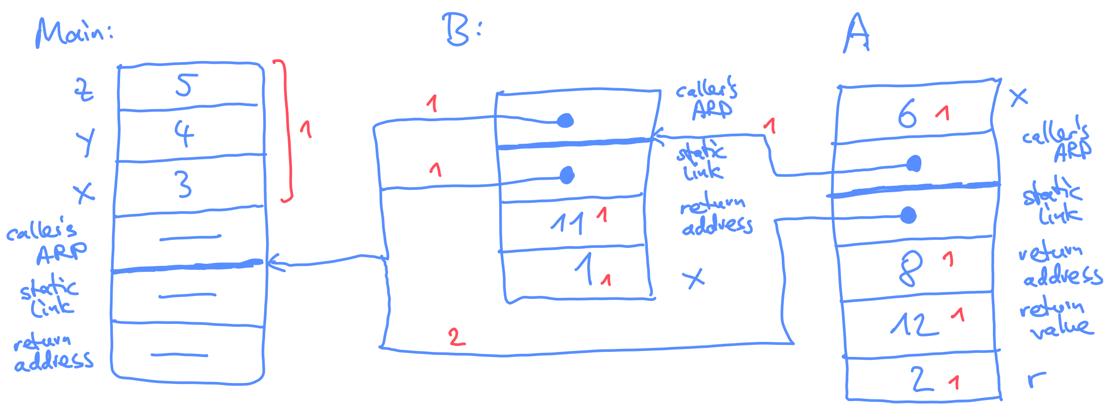
Arrow tips must not be omitted.

We continue with following Pascal program:

```
1  program Main;
2    var x: integer = 3, y: integer = 4, z: integer = 5;
3    function A(r: integer): integer;
4      var x: integer = 6;
5      begin A := r + x + y; end;
6    procedure B(x: integer);
7      begin { body B }
8        A(x + 1);
9      end;
10   begin { Main }
11     B(1);
12   end.
```

c) Draw a graph (as in the lectures) that visualises the structure and contents of the activation records (ARs), including the values of variables, at the following point in the program's execution: In the implementation of line 5, `A` is about to perform the jump back to the return address.

Annotate the ARs' contents (e.g. "return address", "return value", etc.). Draw pointers as arrows. Omit saved registers, and use static links to implement addressability. Write "—" for uninitialised or unknown values. For return addresses, use the line number of the call in the caller's code. Assume a memory-to-memory model and heap-based allocation of ARs. *(11 points)*



Do not give points for unannotated fields.

Cap to 11 points (should have allocated 12 points for this question).

We recall the Pascal program from the previous page:

```
program Main;
  var x: integer = 3, y: integer = 4, z: integer = 5;
  function A(r: integer): integer;
    var x: integer = 6;
    begin A := r + x + y; end;
  procedure B(x: integer);
    begin { body B }
      A(x + 1);
    end;
  begin { Main }
    B(1);
  end.
```

d) Write ILOC code implementing procedure `B` using a memory-to-memory model, heap-based allocation of ARs, and static links. Assume that register `r_hp` points to an area of memory to allocate `A`'s AR in (over increasing memory addresses), that the ILOC code implementing `A` has label `code_A`, and that the callees restore `r_arp` before returning. Do not use symbolic constants for memory offsets—use concrete values. Ignore register saving. Use only the ILOC instructions listed on the last page of this exam booklet. *(11 points)*

```
   loadAI  r_arp, -8 => r_1       // r_1 = x
   addI    r_1,    1 => r_1       // r_1 = x + 1
   store   r_1       => r_hp      // r starts callee's AR
                                  // +4 is return value
   loadI   #r        => r_1
   storeAI r_1       => r_hp, 8   // +8 is return address
   loadAI  r_arp, -4 => r_1
   storeAI r_1       => r_hp, 12  // +12 is static link
   storeAI r_arp     => r_hp, 16  // +16 is caller's AR
   addI    r_hp, 16  => r_arp     // update r_arp
   jumpI             -> code_A    // jump into code for A
                                  // ignore return value
r: loadAI  r_arp, -8 => r_jmp     // load return address
   load    r_arp     => r_arp     // restore caller's ARP
   jump              -> r_jmp     // return
```

Please double-check this self-proclaimed solution, and think about a good and efficient grading scheme for the 11 points!

(This question may be slightly too complex/time-consuming to appear in an exam, but it is good practice for similar but simpler/shorter questions that could appear.)

Question 7. Optimisations (11 points)

The following ILOC code implements an assignment involving variables `a`, `b`, and `c`:

```
loadAI  r_arp, @a  => r_1          // ( 1 –  3)
add     r_1,   r_1 => r_1          // ( 4 –  4)
loadAI  r_arp, @b  => r_2          // ( 5 –  7)
mult    r_1,   r_2 => r_1          // ( 8 –  9)
loadAI  r_arp, @c  => r_2          // (10 – 12)
mult    r_1,   r_2 => r_1          // (13 – 14)
storeAI r_1        => r_arp, @a    // (15 – 17)
```

a) In Java-like syntax, what is the assignment implemented by the ILOC code? *(3 points)*

1 1 1

a = (a + a) * b * c;

or: a = 2*a*b*c; (optimised version)

Let us assume that the ILOC code is executed on an idealised pipelined CPU. On this CPU, at most one instruction can be started in each clock cycle. Instructions `loadAI` and `storeAI` complete in 3 clock cycles, `mult` completes in 2 clock cycles, and `add` completes in 1 clock cycle. An instruction that reads from register r can only start after any previous instruction writing to r is finished. We have annotated the ILOC code above with its execution timing.

b) Reorder the ILOC instructions to implement the same assignment, but take as few clock cycles as possible to execute on the pipelined CPU. You may use different and additional registers. Annotate your optimised code with its execution timing. *(8 points)*

```
loadAI  r_arp, @a  => r_1          // ( 1 – 3 )
loadAI  r_arp, @b  => r_2          // ( 2 – 4 )
loadAI  r_arp, @c  => r_3  ← new register   // ( 3 – 5 )
add     r_1,   r_1 => r_1          // ( 4 – 4 )
mult    r_1,   r_2 => r_1          // ( 5 – 6 )
mult    r_1,   r_3 => r_1          // ( 7 – 8 )
storeAI r_1        => r_arp, @a    // ( 9 – 11 )
```

4.5: → 0 if not the same assignment
→ 2 if same assignment and faster
→ 3 if more than 2 cycles faster
→ 4.5 if optimal

3.5: +0.5 for every correct annotation, taking inherited errors into account

(An ILOC instruction reference table is included on the last page of this exam booklet.)

ILOC Reference

Opcode	Sources	Targets	Meaning	
Memory operations				
loadI	c_1	r_2	$c_1 \Rightarrow r_2$	
load	r_1	r_2	$\text{Memory}(r_1) \Rightarrow r_2$	
loadAI	r_1, c_2	r_3	$\text{Memory}(r_1 + c_2) \Rightarrow r_3$	
loadAO	r_1, r_2	r_3	$\text{Memory}(r_1 + r_2) \Rightarrow r_3$	
store	r_1	r_2	$r_1 \Rightarrow \text{Memory}(r_2)$	
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow \text{Memory}(r_2 + c_3)$	
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow \text{Memory}(r_2 + r_3)$	
Arithmetic				
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$	
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$	
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$	
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$	
mult	r_1, r_2	r_3	$r_1 * r_2 \Rightarrow r_3$	
multI	r_1, c_2	r_3	$r_1 * c_2 \Rightarrow r_3$	
Control flow				
cmp_LT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 < r_2$
			false $\Rightarrow r_3$	otherwise
cmp_LE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \leq r_2$
			false $\Rightarrow r_3$	otherwise
cmp_GT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 > r_2$
			false $\Rightarrow r_3$	otherwise
cmp_GE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \geq r_2$
			false $\Rightarrow r_3$	otherwise
cmp_EQ	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 = r_2$
			false $\Rightarrow r_3$	otherwise
cmp_NE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \neq r_2$
			false $\Rightarrow r_3$	otherwise
cbr	r_1	l_2, l_3	$l_2 \rightarrow$ PC	if r_1 = true
			$l_3 \rightarrow$ PC	otherwise
Jumps				
jumpI	—	l_1	$l_1 \rightarrow$ PC	
jump	—	r_1	$r_1 \rightarrow$ PC	