HOCHSCHULE FÜR TECHNIK **STUTTGART** **SOMMERSEMESTER 2017**

PRÜFUNGSLEISTUNG im STUDIENGANG **S**oftware **T**echnology

MODUL:	CPL	FACH:	Concepts of Programming Languages
DATUM:	15.07.2019	NAME:	
ZEIT:	08:30-10:30	SEMESTER:	ST 1
PRÜFER:	Prof. Dr. Ulrike Padó, Prof. Dr. Peter Heusch		

MODUL-TEILLEISTUNGEN		TEILPUNKTE	GESAMTPUNKTE	NOTE
FACH 1:	CPL	_____	_____	_____

HILFSMITTEL: Original Slides + Notes + Seminar Papers

ANLAGEN:

Please read the following general rules before starting the exam:

- Put your name on every sheet of exam paper that you use
- Do not write final answers on draft paper, do not write final answers in pencil, do not use your own paper
- Write legibly, leave at least 5 mm of empty space (one box) between your lines and at least 40 mm of empty space (8 boxes) at the right margin
- The points for every question reflect the number of minutes needed to answer.

1) Question (30P): Properties of Seminar Languages

Answer the following questions concerning the languages from the seminar. Every question is worth 3 points. Some questions contain a limit on the number of answers. In these cases, giving additional answers results in fewer points!

1) What does it mean for a programming language to be “high-level”? Name two low-level programming languages.
2) Which of the languages are object oriented? Give three names.
3) Which of the languages are object oriented? Give three more names.
4) Which of the languages are purely imperative or procedural? Give three names.
5) What are coroutines? Name two languages that provide coroutines.
6) Name two languages that support pattern matching of code. Which language is used extensively for pattern matching of input?
7) Which languages support type inference? Name three.
8) What are macros? Name two languages that use macros.
9) Which languages are designed for business applications? Give two names. Are they related?
10) If you could start over again, which of the languages in the seminar would you have wanted to learn as your first programming language ever? Why? (80-100 words)

1) High level languages are build for humans to easily understand and program. Low level languages are build for machines.
2) Java, Python, Dart
3) Swift, C, C++
4) Pascal, C, Fortran (imperative
5)A corroutine is a subroutine allowing explicit suspending and resuming at defined points. C, Lua , Kotin7)
8)A macro instruction is a group of programming instructions that have been compressed into a simpler forn and appear as a single instruciotn. Wehen used a macro expands from its comporessed form into its ... Lisp, C,
9)Cobol, abap, adabas

2) Question (30 P): General Concepts in Programming

a) Strong typing and Type Hierarchies: What mythical class does the Java reference literal null belong to, and why is this mythical class necessary? (15)

b) If you wanted to create your own language, would you derive it from another language or implement it from scratch? Since there is no single correct answer, please explain your choices carefully. (15 P)

3) Question (30 P): Formal Grammars and Parsing Algorithms

Look at the following context free grammar that roughly describes lists in Scheme:

List → (Element) | ()

Element → List | Atom | Element Element

Atom → a | b | c | f | g

b)
The language is not LL parsable because the second rule contains left recursion.

Eliminate recursion:
1. List -> (Element) | ()
2. Element -> List Element' | Atom Element'
3. Element' -> Element Element' | epsilon
4. Atom -> a | b | c | f | g

Now the first and follow sets can be compute without a recursion appearing

a) Construct the syntax tree for the input **(f (g a) b).** (10 P)

b) This language is not LL-parseable. Why not? (10 P)

c) How could the grammar be changed to be made LL-parseable? Explain how your changes fix the grammar.
(Hint: empty productions are allowed) (10 P)

4) Question (30 P): Programming Paradigms, Scheme and Prolog

a) Here is a naive implementation of multiplication by addition. Make it tail-recursive and explain what you did. (7 P)

```
(define mult
     (lambda (base factor) (if (= 0 factor)
          0
          (+ base (mult base (- factor 1))))))
```

b) Do you automatically achieve tail recursion if you use an accumulator? Think of the use of the accumulator in Prolog. (7 P)

c) Look at the following Prolog code. What does this predicate do?
Which part is the base case for recursion, which is the recursive call? What happens for the call find(3,[])? Is there an accumulator?
Show the call stack for the call `find(3, [1,2,3]).` (16 P)

```
find(X,[X|List]).
find(X,[Y|List]) :- find(X,List).
```

what does it do?
Recurses down a list to find a membe of the list

the first lin is the base case

the call find(3,[]) fill immediatly return false

Call stack find(3, [1,2,3]).

Call: find(3, [1,2,3,]) ? creep
Call: find(3, [2,3]) ? creep
Call: find(3, [3]) ? creep
Exit: find(3, [3]) ? creep
Exit: find(3, [2,3]) ? creep
Exit: find(3,[1,2,3]) ? creep
true