Formal Semantics of Programming Languages (SS 2019)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The assignments are to be submitted by the deadlines stated below in the following form:

1. A single archive file (zip or tgz format) including
2. a single PDF document with a decent cover page (title of the course, your name, Matrikelnummer, email address) that documents
   - verbal explanations of your results and how to use your programs,
   - descriptions of convincing test runs of the programs,
   - all source code in a structured and easy to read way;
3. all source code files.

Email submissions are not accepted.

You have to submit assignment A and *either* assignment B1 *or* assignment B2 and give a small (5–10 min) presentation of your results for B1/B2.

Assignments B1 and B2 contain an "optional" part that is required (only) if you want to pass the course with grade 1 ("Sehr Gut"); this part can be submitted at a later deadline.

Submission A:	**May 5, 2019**
Submission B1/B2:	**June 23, 2019**
Presentation B1/B2:	**June 28, 2019**
Submission Optional Parts:	**July 21, 2019**

If you lack one of the submissions, contact me for a separate oral exam to replace the submission.

You may choose for your assignments any programming language; programming languages with support for decent high-level data types for the syntactic and semantic domains are recommended. Suitable candidates are typically languages with automatic garbage collection such as Java, C#, Python, Scala, F#, Haskell, Ocaml, . . .

Assignment A: Abstract Syntax Trees and Type Checking

Consider the programming language with procedures introduced in Definitions 7.1 and 7.11 of the manuscript "Thinking Programs". However, rather than having separate syntactic domains for formulas and terms, have a single domain of "expressions" that includes both integer expressions (with integer literals, operations $+$, $*$, unary and binary $-$, $/$, $=$ and $\leq$) and boolean expressions (with truth literals and the logical connectives "and", "or", and "not").

Then perform the following tasks:

1. Adapt/extend the grammar for the abstract syntax of the language and its type system (presented in Figures 7.1 and 7.18) to accommodate your domain of expressions (clearly indicate your changes/extensions).
2. Implement datatypes for the abstract syntax of this language including a function to print out a linear representation of programs.
3. Implement a type checker for this language, i.e., a function that accepts as input the abstract syntax for a program and indicates (by its return value or by throwing an exception) whether the program is well-typed.

See Sections "Abstract Syntax Trees in OCaml" and "Denotational Semantics in OCaml" for inspiration on how to develop such implementations.

Assignment B1: Denotational Semantics of High-Level Programs with Procedures

Consider your programming language with expressions and procedures introduced in Assignment A and perform the following tasks:

- Adapt/extend the denotational semantics for the language presented in Figures 7.2/7.3, 7.6, and 7.20 to accommodate your domain of expressions (clearly indicate your changes respectively extensions).
- Implement your denotational semantics as a function that takes as argument a sequence of values (the initial values of the program parameters) and returns as a result such a sequence (the final values of the program parameters). The function may abort if the evaluation of an expression is not well-defined (e.g., division by zero).
- (Optional) Adapt/extend your type checker to include the translation of variables to addresses depicted in Figure 7.22 by annotating every variable in the syntax tree with an address. Adapt/extend your implementation of the denotational semantics to make use of this annotation as indicated in Figure 7.23.

Assignment B2: Operational Semantics of Machine Programs and Translation of Commands and Expressions

Consider your programming language with expressions introduced in Assignment A (you may ignore procedures) and perform the following tasks:

- Adapt/extend the abstract syntax of machine instructions (introduced in Definition 7.1) to include all machine instructions required for evaluating the expressions of your language (clearly indicate your changes/extensions). Implement a datatype for the abstract syntax of this extended machine language including a function to print out a linear representation.
- Adapt/extend the small-step operational semantics of the machine language introduced in Figure 7.12 to accommodate the additional instructions (clearly indicate your changes respectively extensions). Implement this operational semantics as a function that takes as argument the current configuration of the machine and returns as result the next configuration. The function may abort if the evaluation of an expression is not well-defined (e.g., division by zero). Also implement a function that repeatedly applies the first function as long as the machine configuration indicates that the machine program has not terminated (i.e., the program counter still refers to an instruction in the sequence).
- (Optional) Adapt/extend the translation of expressions and commands introduced in Figures 7.14 and 7.15 to accommodate your extensions. Implement this translation as a function that takes as argument a command and returns as a result a sequence of machine instructions. Test the translation by executing the generated machine program.