Prof. Dr. Peter Thiemann
Matthias Keil Winter Semester 2016/2017

# Compiler Construction

`http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2016ws/`

**Exercise Sheet 3**

## 1 Type-checking MiniJava (5 + 10 Points)

MiniJava is a strongly typed language with explicit types. This means that the type of every variable and every expression is known at compile-time. Detecting type-errors early (i.e. at compile time) supports programmers in writing (fail-)safe code.
MiniJava adheres for the most part to the type rules of Java. It provides two basic types for booleans and integers, and two reference types for integer arrays and objects. Types are defined by

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{int[]} \mid C$$

for all $C \in dom(CT)$ where the class table $CT$ is a mapping from class names to class declarations. There exists a subtype relation $\prec$ between the types. This relation is reflexive and transitive (but not symmetric). For simplicity, we identify the class name with the class type here.

$$\tau \prec \tau \qquad \frac{\tau_1 \prec \tau_2 \qquad \tau_2 \prec \tau_3}{\tau_1 \prec \tau_3} \qquad \frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{ ...\}}}{C \prec D}$$

*Remark:* One major difference between Java and MiniJava is that MiniJava does not specify `Object` to be the superclass of all other classes.

Type judgments define whether an expression, a statement, etc. is *well-typed*. For expressions, we use the type judgment $\Gamma \vdash_e e : \tau$ to say that an expression $e$ is well-typed in $\Gamma$ with type $\tau$. The typing context (or type environment) $\Gamma$ contains all variables with their types which are defined when typing the expression.
For the arithmetic and boolean expressions the type rules are straight-forward, for example:

$$\frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 < e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash_e e : \texttt{bool}}{\Gamma \vdash_e !e : \texttt{bool}}$$

The rules for $-, *$ and && are defined analogously. For constants, the type rules are trivial, as is the one for object allocation:

$$\frac{}{\Gamma \vdash_e \texttt{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash_e \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash_e i : \texttt{int}} \qquad \frac{}{\Gamma \vdash_e \texttt{new}\, C() : C}$$

The type of a variable can be determined by looking it up in the type environment:

$$\frac{id : \tau \in \Gamma}{\Gamma \vdash_e id : \tau}$$

A bit more involved is the type rule for method invocation:

$$\frac{\begin{array}{c}\Gamma \vdash_e e : C \\ paramsT(m, C) = (\tau_1, \ldots, \tau_n) \qquad returnT(m, C) = \tau \qquad \forall e_i : \Gamma \vdash_e e_i : \sigma_i, \quad \sigma_i \prec \tau_i\end{array}}{\Gamma \vdash_e e.m(e_1, \ldots, e_n) : \tau}$$

Here, $paramsT(m, C)$ denotes the types of the formal parameters of method $m$ in class $C$. $returnT(m, C)$ denotes the return type of this method.
Because statements don't have a type, we use a different judgment $\Gamma \vdash_s s$ to denote well-typed statements. The corresponding type rules then have the following form:

$$\frac{\Gamma \vdash_e e : \texttt{int}}{\Gamma \vdash_s \texttt{System.out.println}(e);} \qquad \frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_2 \prec \tau_1}{\Gamma \vdash_s e_1 = e_2;}$$

$$\frac{\Gamma \vdash_e e : \texttt{bool} \qquad \Gamma \vdash_s s}{\Gamma \vdash_s \texttt{while}(e)\ \texttt{do}\ s} \qquad \frac{\Gamma \vdash_e e : \texttt{bool} \qquad \Gamma \vdash_s s_1 \qquad \Gamma \vdash_s s_2}{\Gamma \vdash_s \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2} \qquad \frac{\forall s_i : \Gamma \vdash_s s_i}{\Gamma \vdash_s \{s_1 \ldots s_n\}}$$

Further, a class is well-typed if all its methods are well-typed. A method is well-typed if its statement list is well-typed and the type of its return expression is a subtype of its return type. When type-checking a class or method, `this` must be entered with the correct type in $\Gamma$, as well as all fields of the class and the respective formal parameters and local variables of the method.

## Project - Part 2

- State the type rules for array allocation, array length and lookup in an array. Further, state the type rules for methods and classes.
- Implement a type-checker for MiniJava. Types are represented by the abstract class `Type`. Implement this class together with the subclasses `TInt`, `TIntArray`, `TBool`, and `TClass`. Each class needs to implement the method `boolean isSubtypeOf(Type t)` which checks if the type is a subtype of type `t`.

  The type-checker should proceed in two phases.

  **Phase 1: Building the class table** The class table should contain the following information:

  – each class name is bound to its fields and method declarations;
  – each method name is bound to its parameters, its result type, and local variables;
  – each variable name (for local variables, fields, and parameters) is bound to its type.

Hints and questions:

- How do you deal with superclasses?
- What if a variable or field is declared more than once?
- What about overridden methods?
- Can there be cycles in the type hierarchy?

**Phase 2: Type-checking statements and expressions** Implement now a type-check visitor using the type rules stated above.

- Describe the overall structure of your specification shortly. In particular, explain which steps you take in filling the class table.

---

**Submission**

- Deadline: **08.12.2016, 12:00 (noon)**. Late submissions will not be accepted.
- Submit your solution to the subversion repository. Your submission will consist of one folder (exercise3) which includes your solution.
- A new project template, including a minijava parser, is available on the web.
- Rewrite method `minijava.TypeChecker.typecheckOrFail` so that it calls your type-checker on the given AST.
- Your solution will consist of: 1. a zip file `typechecker.zip` as generated by `ant submission` with the implementation of the type-checker, and 2. a pdf `typechecker-<your name>.pdf` with a description and the type rules.
- You are strongly encouraged to test your solution with the provided test data. Add test cases as you might think necessary. You need not submit your own test cases.
- The description must be limited to two pages. Submitting more than one page will lead to reduction in points.
- The description may be either German or English. Clear and understandable style is required.
- If you want to typeset your solution in LaTeX, you might want to use the `mathpartir` style to set the type rules.