Dr. Annette Bieniusa
M.Sc. Peter Zeller

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lab 3: Compiler and Language Processing Tools (WS 2018)

Deadline: 04.12.2018, 10:00

Name Analysis in Java

In the assignment, you will implement the name and type analysis of a compiler. Besides the important question of *how* such an analysis is implemented, it is of equal importance *what* the analysis is supposed to do. *What* are the rules of name analysis? In contrast to the implementation technique, the rules themselves are language-dependent and much more susceptible to *design choices*.

One of the most important documents for a compiler construction – if not *the* most important one – is the *language specification*. It has to precisely define all the rules for a language. In this exercise we want you to take a look at the Java language specification. The naming rules you are about to implement are those of Java, although MiniJava has much fewer features. The Java language specification (JLS) can be found at `http://docs.oracle.com/javase/specs/`

Here are some example question about name analysis, which you should first try to answer for yourself and then look up in the JLS. You do not have to submit the answer to these questions, but the answers to these questions should help you in implementing name analysis for MiniJava.

- What is a name in Java?
- What is a scope? Is it a static or dynamic property?
- What is the difference between *shadowing*, *obscuring* and *hiding*?
- Can fields and methods of a class have the same name?
- Can a class contain two or more fields of the same name?
- Can a parameter of a method be referenced by a qualified name?
- Can a local variable be declared with the same name as one of the parameters?
- Can a local variable be declared with the same name as one of the fields of the class?
- Can two blocks in the same scope declare local variables with the same name?

Project - Part 2 (14 points)

This exercise requires a working solution of the previous exercise. (*Please contact us, if you did not manage to complete the last exercise. We will then give you a working parser implementation.*) Copy your solution from exercise 2 into a **new folder** named `ex3` and solve this exercise in that new folder.

You can find test-cases for this exercise in `ex3_tests.zip`. The test cases consist of files with type errors in the folder `testdata/typechecker/error` and files without type errors in the folder `testdata/typechecker/ok`. The tests assume that you have a class `Analysis` in package `analysis`, which provides certain methods and constructors. You may adapt the test code to fit to your interface.

1. (4 points) Write code to check if the class hierarchy of a given MiniJava program is consistent. In particular, you should check that:
   a) When a class `extends X`, then `X` must be a class declared in the program.
   b) There are no cycles in the inheritance relation between classes.
   c) Class names, method names, field names, and parameter names in methods are unique.
2. (2 points) Choose a way to represent types in your typechecker and write a method, which checks if one type is a subtype of another type. Explain in your Readme file, how you are representing types.
3. (1 point) Check that methods are overridden correctly, i.e. that the signature is compatible with methods in superclasses with the same name.
4. (7 points) Write code to do type checking and name analysis on the complete program. You can refer to the type rules described at the end of this sheet and to the Java language specification for the details about checking MiniJava.

   Certain information from type checking will be required for the next phase of the project, so make sure that you make at least the following information available for the translation phase:

   - For each `FieldAccess` and `VarUse`: the declaration of the variable.
   - For each `MethodCall`: the declaration of the method.
   - For each `NewObject`: the declaration of the corresponding class.

   Describe in your readme file how you are doing the type checking and how you are going to provide analysis information to the translation phase. In particular explain how you handle different variable scopes.
5. (optional, no points) So far, our type checker performs control-flow insensitive analyses. In contrast, the Java compiler also does control-flow sensitive analyses. For example, in Java a local variable can only be used after it has been initialized. Also return-statements are checked: There cannot be any code after a return statement and every path through the method must have a return statement at the end. Extend your analysis and check these additional constraints. There should be no MiniJava program, which is accepted by your type checker, but rejected by the Java compiler.

   How would you extend the formal type rules to include the new constraints?

Theory (4 points)

4. (2 points) Write a formal type rule for a for-loop statement of the following form:

   `for (int i=e0; e1; s1) s2`

   Here, `i` is an identifier, `e0` and `e1` are expressions, and `s1` and `s2` are statements. A concrete example would be:

   `for (int k=0; k<n; k=k+1) System.out.println(k);`

5. (2 points) Write down a derivation for the following type judgement using the rules at the end of this exercise sheet:

   $$(\textbf{return} : \texttt{int[]}) \vdash_{sl} \texttt{int[] y; y = new int[2]; y[1] = 1; return y;}$$

You can solve this exercise on paper and submit your solution to your repository as pictures (.jpg, .png, or .pdf format). If you want to typeset your solution in LaTeX, you might want to use the `mathpartir` style to set the type rules.

Type rules of MiniJava

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 and enables tool support like sound refactoring and autocompletions.

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. Moreover, there is a special type `null`, which is a subtype of every class type and of the array type, and there is the type $\bot$, which does not have any values. Types are defined as follows:

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

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.

$$\frac{}{\tau \prec \tau}(\textit{refl}) \qquad \frac{\tau_1 \prec \tau_2 \qquad \tau_2 \prec \tau_3}{\tau_1 \prec \tau_3}(\textit{trans}) \qquad \frac{C \in dom(CT)}{\texttt{null} \prec C}(\textit{null-class})$$

$$\frac{}{\texttt{null} \prec \tau[]}(\textit{null-array}) \qquad \frac{CT(C) = \texttt{class } C \texttt{ extends } D \texttt{ \{ ... \}}}{C \prec D}(\textit{extends})$$

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 Γ and has type τ. The typing context (or type environment) Γ is a set containing all fields and local variables with their respective types which are defined when typing the expression. For a local variable x of type τ we have items $(l, x : \tau) \in \Gamma$. For a fields, we write $(f, x : \tau) \in \Gamma$ instead. The environment Γ also contains the return type (we write $(\textbf{return} : \tau) \in \Gamma$) and the current type of **this** (written as $(\textbf{this} : \tau) \in \Gamma$).

For expressions we have the following type rules:

$$\frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 + e_2 : \texttt{int}}(\textit{plus}) \qquad \frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 - e_2 : \texttt{int}}(\textit{minus})$$

$$\frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 * e_2 : \texttt{int}}(\textit{mult}) \qquad \frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 / e_2 : \texttt{int}}(\textit{div})$$

$$\frac{\Gamma \vdash_e e_1 : \texttt{int} \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e e_1 < e_2 : \texttt{bool}}(\textit{less})$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad (\tau_1 \prec \tau_2 \vee \tau_2 \prec \tau_1)}{\Gamma \vdash_e e_1 == e_2 : \texttt{bool}}(\textit{eq}) \qquad \frac{\Gamma \vdash_e e : \texttt{bool}}{\Gamma \vdash_e !e : \texttt{bool}}(\textit{neg})$$

$$\frac{\Gamma \vdash_e e : \texttt{int}}{\Gamma \vdash_e \ -e : \texttt{int}}(\textit{uminus}) \qquad \frac{\Gamma \vdash_e e_1 : \tau[] \qquad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash_e \ e_1[e_2] : \tau}(\textit{array-lookup})$$

$$\frac{\Gamma \vdash_e e : \tau[]}{\Gamma \vdash_e e.\texttt{length} : \texttt{int}}(\textit{array-len}) \qquad \frac{\Gamma \vdash_e e : C \qquad (f, x : \tau) \in \textit{fields}(C)}{\Gamma \vdash_e e.x : \tau}(\textit{field-access})$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e : C \qquad \textit{paramsT}(m, C) = (\tau_1, \ldots, \tau_n) \\ \textit{returnT}(m, C) = \tau \qquad \forall_{i \in \{1,\ldots,n\}} : \ \Gamma \vdash_e e_i : \sigma_i, \quad \sigma_i \prec \tau_i\end{array}}{\Gamma \vdash_e e.m(e_1, \ldots, e_n) : \tau}(\textit{method-call})$$

$$\frac{}{\Gamma \vdash_e \texttt{true} : \texttt{bool}}(\textit{true}) \qquad \frac{}{\Gamma \vdash_e \texttt{false} : \texttt{bool}}(\textit{false}) \qquad \frac{(\_, id : \tau) \in \Gamma}{\Gamma \vdash_e id : \tau}(\textit{var-use})$$

$$\frac{}{\Gamma \vdash_e i : \texttt{int}}(\textit{int-literal}) \qquad \frac{(\textbf{this} : \tau) \in \Gamma}{\Gamma \vdash_e \textit{this} : \tau}(\textit{this}) \qquad \frac{}{\Gamma \vdash_e \texttt{null} : \texttt{null}}(\textit{null})$$

$$\frac{\Gamma \vdash_e e : \texttt{int}}{\Gamma \vdash_e \texttt{new}\ \tau[e][]^n : \tau[][]^n}(\textit{new-array}) \qquad \frac{C \in \textit{dom}(CT)}{\Gamma \vdash_e \texttt{new}\ C() : C}(\textit{new-obj})$$

Because statements do not have a type in MiniJava, we use a different judgment, $\Gamma \vdash_s s$, to denote well-typed statements and $\Gamma \vdash_{sl} s$ for well-typed lists of statements. To handle local variable declarations, we interpret a sequence of statements $s_1; s_2; \ldots s_n$ as being either the empty sequence ϵ or as consisting of one statement followed by a sequence of statements: $s_1; (s_2; \ldots; s_n)$. This way, we can split off the first statement and handle it differently, if it is a variable declaration.

We use the notation "$\Gamma, (\_, x, \tau)$" to denote the updated type environment Γ, were all previous entries for x have been removed and replaced by the mapping $(\_, x, \tau)$.

The corresponding type rules then have the following form:

$$\frac{\Gamma, (l, x : \tau) \vdash_{sl} s \qquad \forall_{\tau'} : \ (l, x : \tau') \notin \Gamma}{\Gamma \vdash_{sl} \tau\ x;\ s}(\textit{var-decl}) \qquad \frac{\Gamma \vdash_s s \qquad \Gamma \vdash_{sl} r}{\Gamma \vdash_{sl} s; r}(\textit{seq})$$

$$\frac{}{\Gamma \vdash_{sl} \epsilon}(\textit{empty}) \qquad \frac{\Gamma \vdash_{sl} s}{\Gamma \vdash_s \{s\}}(\textit{block}) \qquad \frac{\Gamma \vdash_e e : \texttt{bool} \qquad \Gamma \vdash_s s_1 \qquad \Gamma \vdash_s s_2}{\Gamma \vdash_s \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}(\textit{if})$$

$$\frac{\Gamma \vdash_e e : \texttt{bool} \qquad \Gamma \vdash_s s}{\Gamma \vdash_s \texttt{while}(e)\ \texttt{do}\ s}(\textit{while})$$

$$\frac{\Gamma \vdash_e e : \tau_1 \qquad \tau_1 \prec \tau_2 \qquad (\textbf{return} : \tau_2) \in \Gamma}{\Gamma \vdash_s \texttt{return}\ e;}(\textit{return})$$

$$\frac{\Gamma \vdash_e e : \texttt{int}}{\Gamma \vdash_s \texttt{System.out.println}(e);}(\textit{print})$$

$$\frac{\Gamma \vdash_e e : \tau \qquad \text{Expression } e \text{ allowed as statement}}{\Gamma \vdash_s e;}(\textit{stmt-expr})$$

$$\frac{\Gamma \vdash_e e_1 : \tau_1 \qquad \Gamma \vdash_e e_2 : \tau_2 \qquad \tau_2 \prec \tau_1}{\Gamma \vdash_s e_1 = e_2;}(\textit{assign})$$

Further, a class is well-typed if all its methods are well-typed. A method is well-typed if its body is well-typed. When type-checking a class or method, `this` must be entered with the correct type in the typing context Γ, as well as all fields of the class and the respective formal parameters and local variables of the method. These additional conditions are given below:

To increase readability, we use $\overline{x}$ to denote the sequence $x_1, \ldots, x_n$, where all x_i have different names. Similarly, $\overline{(f, x : \tau)}$ stands for $(f, x_1 : \tau_1), \ldots, (f, x_n : \tau_n)$, and so on.

Class typing

$$\frac{(l, p : \bot) \vdash_s s}{\vdash_c \texttt{class}\ C\ \{\texttt{public static void main (String[]}\ p\texttt{)}\ \texttt{s}\ \}}(\textit{main-class})$$

$$\frac{\forall_{m_i \in \overline{m}} : \ \overline{(f, x : \tau)}, (\textbf{this} : \ C) \vdash_m m_i}{\vdash_c \texttt{class}\ C\ \{\overline{\tau\, x};\ \overline{m}\ \}}(\textit{class-no-extends})$$

$$\frac{\forall_{m_i \in \overline{m}} : \ \textit{fields}(C), (\textbf{this} : \ C) \vdash_m m_i \qquad \forall_{m_i \in \overline{m}} : \ \textit{override}(m_i, C, D)}{\vdash_c \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, x};\ \overline{m}\ \}}(\textit{class})$$

Method typing

$$\frac{\Gamma, \overline{(l, p : \tau)}, (\textbf{return} : \tau) \vdash_s s}{\Gamma \vdash_m \tau\ m\ (\overline{\tau\, p})\ s}(\textit{method})$$

Auxiliary definitions

$$\frac{CT(C) = \texttt{class}\ C\ \{\overline{\tau\, x};\ \overline{m}\ \}}{\mathit{fields}(C) = \overline{(f, x : \tau)}}(\textit{fields1})$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, x};\ \overline{m}\ \}}{\mathit{fields}(C) = \mathit{fields}(D), \overline{(f, x : \tau)}}(\textit{fields2})$$

$$\frac{\begin{array}{c}\textit{def}(m, D) \text{ implies} \\ \mathit{returnT}(m, C) \prec \mathit{returnT}(m, D), \quad \mathit{paramsT}(m, C) = \mathit{paramsT}(m, D)\end{array}}{\mathit{override}(m, C, D)}(\textit{override})$$

$$\frac{CT(C) = \texttt{class}\ C\ \{\overline{\tau\, x};\ \overline{m}\ \} \qquad m \text{ defined in } \overline{m}}{\textit{def}(m, C)}(\textit{def1})$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, x};\ \overline{m}\} \qquad m \text{ defined in } \overline{m}}{\mathit{def}(m, C)}(\mathit{def2})$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{\tau\, x};\ \overline{m}\} \qquad m \text{ not defined in } \overline{m} \qquad \mathit{def}(m, D)}{\mathit{def}(m, C)}(\mathit{def3})$$

$$\frac{\mathit{def}(m_i, C) \qquad m_i = \tau\ m\ (\overline{\tau\, p})\ s}{paramsT(m, C) = (\overline{\tau})}(paramsT)$$

$$\frac{\mathit{def}(m_i, C) \qquad m_i = \tau\ m\ (\overline{\tau\, p})\ s}{returnT(m, C) = \tau}(returnT)$$