Dr. Annette Bieniusa
Albert Schimpf, M.Sc.

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Portfolio Exam
Compiler and Language Processing Tools (WS 2020/2021)

General Remarks

Submission

The portfolio for the course "Compiler and Language-processing Tools" in Winter Semester 2020/2021 at TUK will comprise the following artifacts:

- a code repository where you will submit your implementation, and
- a document (pdf) comprising the technical documentation (architecture, design decisions, specific data structures, etc.) and a reflection text (learning process and outcomes)

You have access to a private git repository on our Gitlab server where you can host your portfolio, both code and document. Grading will be done on the last version you pushed before **February 22 2020, 23:59**. We will not take the history of the repository into account when grading. Please commit frequently before the deadline to ensure that your work does not get accidentally deleted or lost.

You can comment your code and submit your documentation and reflection in English or German. Please use the provided template for the document (Word or Latex); add your name to the front page, but do not modify the front page in any other way. By submitting your portfolio for the exam, you confirm the adherence to academic honesty as stated on the front page of the document.

Evaluation

The grading scheme for the implementation is as follows:

Part	**Task**	%
Implementation	*Handling of methods*	10
	Handling of fields and the default object contractors	15
	Simple inheritance (no overriding, no shadowing)	10
	Overriding of methods	5
	Shadowing of fields	5
	Additional test cases	5
	Non-functional requirements	5
Documentation		20
Reflection		25

1 Implementation

The implementation part of the portfolio extends the NotQuiteJava language from the exercises with object-oriented constructs. You have to handle three tasks here:

- Task 1: Name and type analysis
- Task 2: Translation to LLVM
- Task 3: Additional test cases

Code Template Repository

A portfolio code template based on the solution of exercise 4 is provided at `https://pl-git.informatik.uni-kl.de/clp20-groups/clp_solutions`. File test cases for the portfolio task can be found in your portfolio repository under ‘testcases/‘.

Code Style, Documentation, and Linting

Code style is checked automatically with *checkstyle* and follows mostly Google’s Java style guide.[1] All linting options are enabled by default in the provided code template. Please make sure no warnings are generated by the javac compiler.

You should use JavaDoc and comments to document essential aspects of your code. If in doubt, please contact us. We will further take aspects like readability and conciseness into account for the grading.

Task 1: Name and Type Analysis for Object-Oriented Constructs

In Task 1, you will extend the name and type analysis for object-oriented constructs of NotQuiteJava.

As for the exercises, you can refer to the Java language specification (JLS) [2] for details and clarification on the specification.

Here are some questions related to the name analysis that should help you to identify the challenges in implementing the name analysis and guide you in the design and implementation of the type checker. You do not have to submit the answer to these questions.

- 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?

[1]`https://google.github.io/styleguide/javaguide.html`
[2]`http://docs.oracle.com/javase/specs/`

Hints:

1. You can find test-cases for this exercise in the portfolio task in the materials repository.

   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.

2. 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, how you are representing types.

3. Check that methods are overridden correctly, i.e. that the signature is compatible with methods in superclasses with the same name.

4. Write code to check if the class hierarchy of a given NotQuiteJava 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[3].

   d) Global static function names are unique.

5. 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 NotQuiteJava.

   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, FunctionCall`: the declaration of the function.
   - For each `NewObject`: the declaration of the corresponding class.

   Describe in your technical documentation how you are doing the type checking and how you are going to provide analysis information to the translation phase. In particular, you have to explain how you handle different variable scopes.

6. In contrast to the lecture material, you need to handle only the implicit default constructor; you need to cover neither custom constructors nor `super` as part of the portfolio.

Task 2: Translation of Object-Oriented Constructs

Extend the NotQuiteJava-to-MiniLLVM translator with the translation of object-oriented constructs. With this step, you will obtain a complete compiler for the NotQuiteJava language.

[3]There is no overloading in NotQuiteJava.

Hints:

1. You can refer to the LLVM documentation of exercise sheet 4.
2. Translate method calls to procedure calls, where the address of the procedure is read from the virtual method table of the receiver object.
3. Generate a struct type for every class. The struct type should have a reference to the virtual method table as the first field. Additionally, it should include fields for all the fields from the Java class. Remember that the order of fields is important and that the memory layout of classes has to be compatible with the memory layout of their super-classes.
4. For representing virtual method tables, you can create a struct type and a corresponding global, constant variable. The fields of the virtual method tables are pointers to procedures, so you have to calculate a corresponding type.
5. Create a procedure for the constructor of each class. In the constructor, you should allocate memory for the object (`alloc` instruction) and initialize it. You also have to set the reference to the virtual method table.
6. You can use the `SizeOf` operand to calculate the size for instances of a struct-type.
7. Remember that objects can be `null` in Java.
8. There is no subtyping between struct types in LLVM, so you will have to use casts (`bitcast` instruction) in places where Java allows subtyping (e.g. assignments, equality checks, method calls, return statements).

Task 3: Additional tests

You should add up at least 5 additional file test cases **for the type checking that yield type errors** and at least 5 additional file test cases **for the translation** that complement the tests given in the template. Please ensure that the tests are minimal and concentrate on one aspect each. For each test, you need to provide a short description to explain what aspect the tests covers.

Hints:

- You can submit the tests, even if you do not implement the functionality required to pass them.
- Please prepend the file names for your test cases with `portfolio`, e.g. `portfolio_test1.java`, `portfolio_test2.java` etc.

2 Documentation

The documentation comprises a summary of your portfolio work and a description of the algorithms and data structures that you used in your implementation. In the text, you should explain and justify your design decisions. Please use graphics and sketches as you see fit.

You should check Google's Design Document Procedure[4] regarding style and structure of your documentation. We suggest having a look at the following sections:

- General principles: `https://developers.google.com/style/tone`

[4] `https://developers.google.com/style`

- Computer interfaces: `https://developers.google.com/style/api-reference-comments`

Please use the template document for submitting the portfolio (as pdf, approx. 5-10 pages, 2.000 - 2.500 words).

3 Reflection

The purpose of the reflection is to show that you are able to reflect and assess your own work and learning process critically. It should answer the following questions:

- What was the most interesting thing that you learned while working on the portfolio? What aspects did you find interesting or surprising?
- Which part of the portfolio are you (most) proud of? Why?
- What adjustments to your design and implementation were necessary during the implementation phase? What would you change or do different if you had to do the portfolio task a second time?
- From the lecture/course, what topic did excite you most? Why? What would you like to learn more about and why?

You may add further aspects as you see fit; if in doubt, please contact us for advise.

Please use the template document for submitting the portfolio (as pdf, approx. 3-5 pages, 1.500 - 2.000 words).

Type rules of NotQuiteJava

NotQuiteJava 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 autocompletion.

NotQuiteJava adheres for the most part to the type rules of Java. It provides two basic types for booleans and integers, and reference types for objects and arrays. Moreover, there is a special type `null`, which is a subtype of every class type and of the array type. Types are defined as follows:

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

for all $C \in \mathit{dom}(CT)$ 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.

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

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

Remark: One major difference between Java and NotQuiteJava is that NotQuiteJava 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:

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

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

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

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

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

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

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

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

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

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

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

Because statements do not have a type in NotQuiteJava, 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:

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

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

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

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

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

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

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

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

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

Method and function 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

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

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

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

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

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

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

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

$$\frac{m_i = \tau\ m\ (\overline{\tau\, p})\ s \qquad m_i \text{ defined globally}}{\textit{paramsT}(m) = (\overline{\tau})}(\textit{paramsT-function})$$

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

$$\frac{m_i = \tau\ m\ (\overline{\tau\, p})\ s \qquad m_i \text{ defined globally}}{returnT(m) = \tau}\,(\textit{returnT-function})$$