École Polytechnique 2024–2025

`CSC_52064` Compilation

Mini Java

version 1 — January 25, 2025

The goal is to build a compiler for a tiny fragment of the Java language, called Mini Java in the following, to x86-64 assembly. This fragment contains integers, Booleans, strings, and objects. It is compatible with Java. This means that Java can be used as a reference when needed.

The syntax of Mini Java is described in Sec. 1. A parser is provided (for both OCaml and Java). You have to implement static type checking (Sec. 2) and code generation (Sec. 3).

1 Syntax

We use the following notations in grammars:

$\langle rule \rangle^\star$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero)
$\langle rule \rangle_t^\star$	repeats $\langle rule \rangle$ an arbitrary number of times (including zero), with separator t
$\langle rule \rangle^+$	repeats $\langle rule \rangle$ at least once
$\langle rule \rangle_t^+$	repeats $\langle rule \rangle$ at least once, with separator t
$\langle rule \rangle?$	use $\langle rule \rangle$ optionally
($\langle rule \rangle$)	grouping

Be careful not to confuse "*" and "$^+$" with "`*`" and "`+`" that are Java symbols. Similarly, do not confuse grammar parentheses with terminal symbols `(` and `)`.

1.1 Lexical Conventions

Spaces, tabs, and newlines are blanks. Comments are of two kinds:

- delimited by `/*` and `*/` (and not nested);
- starting from `//` and extending to the end of line.

Identifiers follow the regular expression $\langle ident \rangle$:

$$\begin{array}{lll} \langle digit \rangle & ::= & \texttt{0–9} \\ \langle alpha \rangle & ::= & \texttt{a–z} \mid \texttt{A–Z} \\ \langle ident \rangle & ::= & (\langle alpha \rangle \mid \texttt{\_})\ (\langle alpha \rangle \mid \texttt{\_} \mid \langle digit \rangle)^\star \end{array}$$

The following identifiers are keywords:

```
boolean     class    else    extends    false    for     if
instanceof  int      new     null       public   return  static
this        true     void
```

Integer literals follow the regular expression ⟨*integer*⟩:

$$\langle \textit{integer} \rangle \quad ::= \quad \texttt{0} \mid \texttt{1–9}\ \langle \textit{digit} \rangle^\star$$

String literals are written between quotes (`"`). There are three escape sequences: `\"` (for the character `"`), `\n` (for a newline character), and `\\` (for the character `\`).

1.2 Syntax

The grammar of source files is given in Fig. 1 and Fig. 2. The entry point is ⟨*file*⟩. Associativity and priorities are given below, from lowest to strongest priority.

operation	**associativity**	**priority**
`=`	right	lowest
`||`	left	
`&&`	left	
`==`, `!=`	left	
`>`, `>=`, `<`, `<=`, `instanceof`	left	↓
`+`, `-`	left	
`*`, `/`, `%`	left	
`-` (unary), `!`, cast	right	
`.`	left	strongest

⟨*file*⟩ ::= ⟨*class*⟩* ⟨*class_Main*⟩ `EOF`
⟨*class*⟩ ::= `class` ⟨*ident*⟩ (`extends` ⟨*ident*⟩)? `{` decl* `}`
⟨*decl*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ `;` | ⟨*constructor*⟩ | ⟨*method*⟩

⟨*constructor*⟩ ::= ⟨*ident*⟩ `(` ⟨*params*⟩? `)` `{` ⟨*stmt*⟩* `}`
⟨*method*⟩ ::= (⟨*type*⟩ | `void`) ⟨*ident*⟩ `(` ⟨*params*⟩? `)` `{` ⟨*stmt*⟩* `}`

⟨*params*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ | ⟨*type*⟩ ⟨*ident*⟩ `,` ⟨*params*⟩
⟨*type*⟩ ::= `boolean` | `int` | ⟨*ident*⟩

⟨*class_Main*⟩ ::=
```
class Main {
public static void main(String ⟨ident⟩ []) { ⟨stmt⟩* }
}
```

Figure 1: Grammar of Mini Java (files).

Syntactic Sugar.

- `if (`e_1`)` e_2 is sugar for `if (`e_1`)` e_2 `else;`.
- A call `m(e1,...,e2)` is sugar for `this.m(e1,...,e2)`.
- In a loop `for (e1;e2;e3)`, the expression `e2` is `true` when omitted.

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle integer \rangle \mid \langle string \rangle \mid \texttt{true} \mid \texttt{false} \\
& \mid & \texttt{this} \\
& \mid & \texttt{null} \\
& \mid & \texttt{(}\ \langle expr \rangle\ \texttt{)} \\
& \mid & \langle ident \rangle \\
& \mid & \langle expr \rangle\ \texttt{.}\ \langle ident \rangle \\
& \mid & \langle ident \rangle\ \texttt{=}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \texttt{.}\ \langle ident \rangle\ \texttt{=}\ \langle expr \rangle \\
& \mid & \langle ident \rangle\ \texttt{(}\ \langle lexpr \rangle ?\ \texttt{)} \\
& \mid & \langle expr \rangle\ \texttt{.}\ \langle ident \rangle\ \texttt{(}\ \langle lexpr \rangle ?\ \texttt{)} \\
& \mid & \texttt{new}\ \langle ident \rangle\ \texttt{(}\ \langle lexpr \rangle ?\ \texttt{)} \\
& \mid & \texttt{!}\ \langle expr \rangle \\
& \mid & \texttt{-}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \langle binop \rangle\ \langle expr \rangle \\
& \mid & \texttt{(}\ \langle type \rangle\ \texttt{)}\ \langle expr \rangle \\
& \mid & \langle expr \rangle\ \texttt{instanceof}\ \langle type \rangle \\
\\
\langle binop \rangle & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \\
\langle lexpr \rangle & ::= & \langle expr \rangle \mid \langle expr \rangle\ \texttt{,}\ \langle lexpr \rangle \\
\\
\langle stmt \rangle & ::= & \texttt{;} \\
& \mid & \langle expr \rangle\ \texttt{;} \\
& \mid & \langle type \rangle\ \langle ident \rangle\ \texttt{;} \\
& \mid & \langle type \rangle\ \langle ident \rangle\ \texttt{=}\ \langle expr \rangle\ \texttt{;} \\
& \mid & \texttt{if (}\ \langle expr \rangle\ \texttt{)}\ \langle stmt \rangle \\
& \mid & \texttt{if (}\ \langle expr \rangle\ \texttt{)}\ \langle stmt \rangle\ \texttt{else}\ \langle stmt \rangle \\
& \mid & \texttt{for (}\ \langle expr \rangle ?\ \texttt{;}\ \langle expr \rangle ?\ \texttt{;}\ \langle expr \rangle ?\ \texttt{)}\ \langle stmt \rangle \\
& \mid & \texttt{\{}\ \langle stmt \rangle^{\star}\ \texttt{\}} \\
& \mid & \texttt{return}\ \langle expr \rangle ?\ \texttt{;}
\end{array}
$$

Figure 2: Grammar of Mini Java (expressions and statements).

2 Static Typing

Static types τ are given by the following grammar:

$$\tau ::= \texttt{void} \mid \texttt{boolean} \mid \texttt{int} \mid C \mid \texttt{typenull}$$

where C is a class. It is convenient to consider `void` as a type, even if it is not a type in the syntax. Beside, `typenull` is introduced to give a type to `null`. We say that a type τ is well formed, and we write τ *wf*, if it is either `boolean`, or `int`, or *Object*, or *String*, or a class C declared in the source file.

Inheritance and Subtyping. We note $C_1 \longrightarrow C_2$ the relation "the class C_1 is a subclass of class C_2", which is the reflexive-transitive closure of the `extends` declarations.

There are two predefined classes: *Object* and *String*. When a class does not inherit from another class with `extends`, it implicitly inherits from *Object*. The class *String* inherits from *Object*. The class *Object* does not inherit from any other class.

The subtyping relation $\tau_1 \sqsubseteq \tau_2$ means "the type τ_1 is a subtype of type τ_2" and is defined as follows:

$$\frac{\tau \in \{\texttt{boolean},\ \texttt{int}\}}{\tau \sqsubseteq \tau} \qquad \frac{C_1 \longrightarrow C_2}{C_1 \sqsubseteq C_2} \qquad \frac{}{\texttt{typenull} \sqsubseteq C}$$

We can interpret $\tau_1 \sqsubseteq \tau_2$ as "any value of type τ_1 can be used when is value of type τ_2 is expected". We say that types τ_1 and τ_2 are *compatible*, and we write $\tau_1 \equiv \tau_2$, if $\tau_1 \sqsubseteq \tau_2$ or $\tau_2 \sqsubseteq \tau_1$. Subtyping extends to lists of types as follows:

$$(\tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_1, \ldots, \tau'_n) \text{ if and only if } \tau_i \sqsubseteq \tau'_i \text{ for all } i \in 1, \ldots n.$$

2.1 Attributes, Constructors, and Methods

We write $C\{\ \tau\ x\ \}$ the fact that class C contains an attribute x of type τ. This attribute is either declared in class C, or inherited from the super-class of C.

We write $C\{\ C(\tau_1, \ldots, \tau_n)\ \}$ the fact that class C has a constructor with type $C(\tau_1, \ldots, \tau_n)$. In Mini Java, each class has **exactly one constructor**. (There is no overloading of constructors.) When no constructor is explicitly declared, an implicit constructor with no parameters is assumed.

We write $C\{\ \tau\ m(\tau_1, \ldots, \tau_n)\ \}$ the fact that class C has a method m with parameters of types $\tau_1, \ldots, \tau_n$ and return type τ. This method is either declared in class C, or inherited from the super-class of C. In Mini Java, each class has **at most one method** with a given name m. (There is no overloading of methods.)

2.2 Typing Rules for Expressions

In the following, C_0 stands for the current class, that is the class in which we are currently performing type checking.

A typing environment Γ is a sequence of variable declarations $\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n$. It is used only for local variables, parameters of constructors and methods, and `this`. The

judgment $\Gamma \vdash e : \tau$ means "in environment Γ, expression e is well typed of type τ". It is defined as follows:

$$\frac{c \text{ constant of type } \tau}{\Gamma \vdash c : \tau} \qquad \frac{}{\Gamma \vdash \texttt{null} : \texttt{typenull}} \qquad \frac{C\ \texttt{this} \in \Gamma}{\Gamma \vdash \texttt{this} : C}$$

$$\frac{\tau\ x \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{x \notin \Gamma \quad C_0\{\ \tau\ x\ \}}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : C \quad C\{\ \tau\ x\ \}}{\Gamma \vdash e.x : \tau}$$

$$\frac{\tau_1\ x \in \Gamma \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash x = e_2 : \tau_1} \qquad \frac{x \notin \Gamma \quad C_0\{\ \tau_1\ x\ \} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash x = e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : C \quad C\{\ \tau_1\ x\ \} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e_1.x = e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}\ e : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{boolean}}{\Gamma \vdash \texttt{!}e : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 \equiv \tau_2 \quad op \in \{\texttt{==},\ \texttt{!=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{<},\ \texttt{<=},\ \texttt{>},\ \texttt{>=}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad op \in \{\texttt{+},\ \texttt{-},\ \texttt{*},\ \texttt{/},\ \texttt{\%}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{boolean} \quad \Gamma \vdash e_2 : \texttt{boolean} \quad op \in \{\texttt{\&\&},\ \texttt{||}\}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \in \{\texttt{int},\ \texttt{String}\}}{\Gamma \vdash e_1 + e_2 : \texttt{String}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \texttt{String} \quad \tau_1 \in \{\texttt{int},\ \texttt{String}\}}{\Gamma \vdash e_1 + e_2 : \texttt{String}}$$

$$\frac{\Gamma \vdash e : \texttt{String}}{\Gamma \vdash \texttt{System.out.print}(e) : \texttt{void}} \qquad \frac{\Gamma \vdash e_1 : \texttt{String} \quad \Gamma \vdash e_2 : \texttt{String}}{\Gamma \vdash e_1.\texttt{equals}(e_2) : \texttt{boolean}}$$

$$\frac{\Gamma \vdash e : C \quad \Gamma \vdash e_i : \tau_i \quad C\{\ \tau\ m(\tau'_1, \ldots, \tau'_n)\ \} \quad \forall i,\ \tau_i \sqsubseteq \tau'_i}{\Gamma \vdash e.m(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \quad C\{\ C(\tau'_1, \ldots, \tau'_n)\ \} \quad \forall i,\ \tau_i \sqsubseteq \tau'_i}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C}$$

$$\frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau'}{\Gamma \vdash (\tau)e : \tau} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau \equiv \tau' \quad \tau' \in \{C, \texttt{typenull}\}}{\Gamma \vdash e\ \texttt{instanceof}\ \tau : \texttt{boolean}}$$

2.3 Typing Rules for Statements

The judgment $\Gamma \vdash s \to \Gamma'$ means "in environment Γ, the statement s is well typed and defines a new environment Γ'". It is defined as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e\texttt{;} \to \Gamma} \qquad \frac{x \notin \Gamma \quad \tau \ wf}{\Gamma \vdash \tau\ x\texttt{;} \to \Gamma, \tau\ x} \qquad \frac{x \notin \Gamma \quad \tau \ wf \quad \Gamma \vdash e : \tau' \quad \tau' \sqsubseteq \tau}{\Gamma \vdash \tau\ x\ \texttt{=}\ e\texttt{;} \to \Gamma, \tau\ x}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \quad \Gamma \vdash s_1 \to \Gamma_1 \quad \Gamma \vdash s_2 \to \Gamma_2}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \to \Gamma}$$

$$\frac{\Gamma \vdash e_1\texttt{;} \to \Gamma \quad \Gamma \vdash e_2 : \texttt{boolean} \quad \Gamma \vdash e_3\texttt{;} \to \Gamma \quad \Gamma \vdash s \to \Gamma_1}{\Gamma \vdash \texttt{for(}e_1\texttt{;}\ e_2\texttt{;}\ e_3\texttt{)}\ s \to \Gamma}$$

$$\frac{}{\Gamma \vdash \texttt{;} \to \Gamma} \qquad \frac{\Gamma \vdash s_1 \to \Gamma_1 \quad \Gamma_1 \vdash s_2 \to \Gamma_2}{\Gamma \vdash s_1\texttt{;}\ s_2 \to \Gamma_2}$$

$$\frac{\Gamma \vdash i \to \Gamma'}{\Gamma \vdash \texttt{\{}i\texttt{\}} \to \Gamma} \qquad \frac{}{\Gamma \vdash \texttt{return;} \to \Gamma} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e\texttt{;} \to \Gamma}$$

2.4 Typing Rules for Classes

2.4.1 Existence and Uniqueness

To be well typed, a file must satisfy the following constraints:

- each class is defined only once;
- a class must inherit from an existing class, different from `String`;
- the inheritance relation must not contain a cycle.

Classes can appear in any order. At any point we can refer to a class which is declared later in the file. Other constraints are as follows:

- attributes of a given class must be distinct;
- each class has at most one constructor (no overloading);
- each class has at most one method of a given name (no overloading).

Overriding. If a method m in class C is overridden is class C', then it must have the same type parameters and the same return type in both classes.

2.4.2 Typing Rules for Attributes, Constructors, and Methods

Let C_0 be the current class. The initial typing environment is $\Gamma_0 = C_0$ `this`.

Typing Attributes. For the declaration of an attribute $\tau\ x$, the type τ must be well formed.

Typing Constructors. A constructor C_0`(`$\tau_1\ x_1, \ldots, \tau_n\ x_n$`){`$s$`}` if well typed if identifiers x_i are pairwise distinct, if all types τ_i are well formed, and if the block s is well typed in the environment $\Gamma_0, \tau_1\ x_1, \ldots, \tau_n\ x_n$.

Typing Methods. A method τ m`(`$\tau_1\ x_1, \ldots, \tau_n\ x_n$`){`$s$`}` is well typed if all identifiers x_i are pairwise distinct, if all types τ_i are well formed, and if the block s is well typed in the environment $\Gamma_0, \tau_1\ x_1, \ldots, \tau_n\ x_n$.

Beside, any occurrence of `return` in s must return a value of a subtype of τ. Finally, when τ is not `void`, any execution flow is s must contain a `return` statement.

2.5 Hints

It is strongly advised to proceed in three steps:

1. declare all classes and check for uniqueness of classes;
2. declare inheritance relations (`extends`) attributes, constructors, and methods;
3. type check the body of constructors and methods.

3 Code Generation

Will be given later.

4 Project Assignment (due March 16, 6pm)

The project must be done **alone or in pair, in Java or OCaml**. It must be delivered on Moodle, as a compressed archive containing a directory with your name(s) (*e.g.* `dupont-durand`). Inside this directory, source files of the compiler must be provided (no need to include compiled files). The command `make` must create the compiler, named `minijava`. The compilation may involve any tool (such as `dune` for OCaml) and the `Makefile` can be as simple as a call to such a tool. The command `minijava` may be a script to run the compiler, for instance if the compiler is implemented in Java.

The archive must also contain **a short report** explaining the technical choices and, if any, the issues with the project and the list of whatever is not delivered. The report can be in format ASCII, Markdown, or PDF.

The command line of `minijava` accepts an option (among `--parse-only` and `--type-only`) and exactly one file with extension `.java`. If the file is parsed successfully, the compiler must terminate with code 0 if option `--parse-only` is on the command line. Otherwise, the compiler moves to static type checking. Any type error must be reported as follows:

```
file.java:4:6:
bad arity for method m
```

The location indicates the filename name, the line number, and the column number. Feel free to design your own error messages. The exit code must be 1.

If the file is type-checked successfully, the compiler must exit with code 0 if option `--type-only` is on the command line. Otherwise, the compiler generates x86-64 assembly code in file `file.s` (same name as the input file, but with extension `.s` instead of extension `.java`). The x86-64 file will be compiled and run as follows

```
gcc file.s -o file
./file
```

possibly with option `-no-pie` on the `gcc` command line. Any runtime error must be reported, but no location nor a detailed message is expected so it is fine to simply output

```
error
```

and terminate with exit code 1.