# Type-checking with Grace

Kim B. Bruce* — Pomona College
Andrew Black — Portland State University
James Noble, Michael Homer — Victoria University of Wellington

February 22, 2023

## 1 Preliminary Definitions

**Definition 1.1** *Let $\mathcal{V}$ be an infinite collection of type variables, $\mathcal{L}$ be an infinite collection of labels, and $\mathcal{C}$ be a collection of type constants that includes at least the type constants Boolean, Number,* `Object`*, and* `Done`*. The simple pre-type expressions, PreType, and record pre-type expressions, PreRecType, of Grace with respect to $\mathcal{V}$, $\mathcal{L}$, and $\mathcal{C}$ are given by the following context-free grammar. We assume $t, t_i \in \mathcal{V}$, $c \in \mathcal{C}$, and $m_i \in \mathcal{L}$ in the following.*

$$\tau \in \mathit{PreType} ::= t \mid c \mid \texttt{ref}\ \tau \mid \tau_1 \times \ldots \times \tau_n \rightarrow \tau \mid \forall t_1 <: \tau_1, \ldots, t_n <: \tau_n.\tau \mid \\ \{m_1 \colon \tau_1; \ldots; m_n \colon \tau_n\} \mid\ < l_1 \colon \tau_1, \ldots, l_n \colon \tau_n >$$

Type `Object` will stand for the type of an imperative command expression, i.e., an expression that does not return a value. The type `Object` is a supertype of all object types, and (in our current implementation) contains `asString` and `asDebugString` methods that are (implicitly) inherited by all object types.

Reference types are the types of variables. That is, if $x$ is a variable holding values of type $\tau$, then $x$ has type `ref` $\tau$. This notation allows us to distinguish between values of type $\tau$ and variables that hold values of that type.

As is standard, the type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is the type of functions taking parameters of type $\tau_1$ through $\tau_n$ and returning a value of type $\tau$. The type $\forall t_1 <: \tau_1, \ldots, t_n <: \tau_n.\tau$ represents bounded polymorphic functions (that is functions that take types as parameters). Unbounded polymorphic functions can be represented by terms of the form $\langle t_1 <: \texttt{Object}, \ldots, t_n <: \texttt{Object}\rangle.\tau$. The identifiers $t_i$ are bound by these type expressions. As usual we identify polymorphic types that are the same up to renaming of the bound variable.

The type $\{m_1 \colon \tau_1; \ldots; m_n \colon \tau_n\}$ represents the type of an object with public methods $m_1, \ldots, m_n$. On the other hand the type $< l_1 \colon \tau_1, \ldots, l_n \colon \tau_n >$ represents a tagged variant type. A typical element, written $< l_i = e_i >$, has the type if $e_i$ has type $\tau_i$

We will use the abbreviation $\texttt{Block0}[\![, \sigma]\!]$ to stand for the type $\{\mathit{apply} \colon \rightarrow \sigma\}$, $\texttt{Block1}[\![\tau, \sigma]\!]$ to stand for the type $\{\mathit{apply} \colon \tau \rightarrow \sigma\}$, etc.

The axioms and rules for determining valid types and constructors are given with respect to a set, $C$, of simple type constraints, which provide information about free type variables. The definition

*Corresponding author: Kim B. Bruce, Department of Computer Science, Pomona College, Claremont, CA 91711. kim@cs.pomona.edu

of type constraints, the rules for determining valid types and constructors, and the matching and subtyping rules are mutually recursive.

**Definition 1.2** *Relations of the form $t <: \tau$ and $t = \tau$, where $t$ is a type variable and $\tau$ is a type expression, are said to be* simple type constraints*. A* type constraint system*, $C$, is defined as follows:*

1. *The empty set, $\emptyset$, is a type constraint system.*
2. *If $C$ is a type constraint system such that $C \vdash \tau$:* `TYPE`*, and $t$ is a type variable that does not appear in $C$, then $C \cup \{t <: \tau\}$ is a type constraint system.*
3. *If $C$ is a type constraint system such that $C \vdash \tau <:$* `Object`*, and $t$ is a type variable that does not appear in $C$ or $\tau$, then $C \cup \{t = \tau\}$ is a type constraint system.*

In Figure 1 we include axioms and rules for determining which are the legal type and constructor expressions. In the `ObjectType` rule, the types of all methods must be function or polymorphic function types.

The axioms and rules for $<:$ can be found in Figure 2. Reference types have no non-trivial subtypes. The subtyping rule for function types is contravariant in the argument type and covariant in the result type. The subtyping rules for polymorphic types support covariant changes in the return type, but no changes in the bounds of the type parameters. While the rules could be generalized to allow contravariant changes in the type bounds, the decidability of subtyping would be lost. The subtyping rule for object types allows both depth and width subtyping.

*Note that Pierce, pg 197, uses different rules.*

$$C \vdash c\colon \texttt{TYPE},\ \ \textit{for}\ c \in \mathcal{C}$$

$$C \cup \{t <: \tau\} \vdash\ \ t\colon \texttt{TYPE}$$

$$C \cup \{t = \tau\} \vdash\ \ t\colon \texttt{TYPE}$$

$$\frac{C \vdash \sigma_i\colon \texttt{TYPE}\ \textit{for}\ i = 1, \ldots, n \qquad C \vdash \tau\colon \texttt{TYPE}}{C \vdash \sigma_1 \times \ldots \times \sigma_n \to \tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{t <: \gamma\} \vdash \tau\colon \texttt{TYPE}}{C \vdash \forall t <: \gamma.\tau\colon \texttt{TYPE}}$$

$$\frac{C \vdash \tau_i\colon \texttt{TYPE}\ \textit{for}\ 1 \leq n}{C \vdash \{l_i\colon \tau_i\}_{i \leq n}\colon \texttt{TYPE}}$$

$$\frac{C \vdash \tau_i\colon \texttt{TYPE}\ \textit{for}\ 1 \leq n}{C \vdash < l_i\colon \tau_i >_{i \leq n}\colon \texttt{TYPE}}$$

$$\frac{C \vdash \tau\colon \texttt{TYPE}}{C \vdash \texttt{ref}\ \tau\colon \texttt{TYPE}}$$

$$\frac{C \cup \{t\colon \texttt{TYPE}\} \vdash \kappa\colon K}{C \vdash \texttt{TFunc}[t]\ \kappa\colon \texttt{TYPE} \Rightarrow K}$$

$$\frac{C \vdash \kappa\colon \texttt{TYPE} \Rightarrow K, \quad C \vdash \tau\colon \texttt{TYPE}}{\kappa[\tau]\colon K}$$

Figure 1: Type and constructor rules

*Refl*$_{<:}$ $$\frac{C \vdash \tau\colon \mathtt{TYPE}}{C \vdash \tau <: \tau}$$

*Hyp*$_{<:}$ $$C \cup \{t <: \tau\} \vdash t <: \tau$$

*Trans*$_{<:}$ $$\frac{C \vdash \sigma <: \tau \qquad C \vdash \tau <: \delta}{C \vdash \sigma <: \delta}$$

*Func*$_{<:}$ $$\frac{C \vdash \sigma <: \sigma' \qquad C \vdash \tau' <: \tau}{C \vdash \sigma' \rightarrow \tau' <: \sigma \rightarrow \tau}$$

*Poly*$_{<:}$ $$\frac{C \cup \{u\colon \mathtt{TYPE}\} \vdash \tau'[t' \mapsto u] <: \tau[t \mapsto u] \qquad u \notin FV(\tau') \cup FV(\tau)}{C \vdash \forall t'.\tau' <: \forall t.\tau}$$

*SBdedPoly*$_{<:}$ $$\frac{C \cup \{u <: \gamma\} \vdash \tau'[t' \mapsto u] <: \tau[t \mapsto u] \qquad u \notin FV(\tau') \cup FV(\tau) \cup FV(\gamma)}{C \vdash \forall t' <: \gamma.\tau' <: \forall t <: \gamma.\tau}$$

*ObjType*$_{<:}$ $$\frac{C \vdash \tau_i' <: \tau_i \quad for\ 1 \leq i \leq n, \qquad C \vdash \tau_i'\colon \mathtt{TYPE} \quad for\ n+1 \leq i \leq n+m}{C \vdash \{l_j\colon \tau_j'\}_{j \leq n+m} <: \{l_j\colon \tau_j\}_{j \leq n}}$$

*VarWidth*$_{<:}$ $$\frac{C \vdash \tau_i' <: \tau_i \quad for\ 1 \leq i \leq n, \qquad C \vdash \tau_i'\colon \mathtt{TYPE} \quad for\ n+1 \leq i \leq n+m}{C \vdash < l_j\colon \tau_j' >_{j \leq n} <: < l_j\colon \tau_j >_{j \leq n+m}}$$

*VariantLeft*$_{<:}$ $$\frac{C \vdash \sigma <: \gamma \quad C \vdash \tau <: \gamma}{C \vdash \sigma | \tau <: \gamma}$$

*VariantRight*$_{<:}$ $$\frac{C \vdash \gamma <: \sigma \quad or \quad C \vdash \gamma <: \tau}{C \vdash \gamma <: \sigma | \tau}$$

*AndLeft*$_{<:}$ $$\frac{C \vdash \sigma <: \gamma \quad or\ C \vdash \tau <: \gamma}{C \vdash \sigma \& \tau <: \gamma}$$

*AndRight*$_{<:}$ $$\frac{C \vdash \gamma <: \sigma, \quad C \vdash \gamma <: \tau}{C \vdash \gamma <: \sigma \& \tau}$$

Figure 2: Subtyping rules

## 1.1 Grace Expression Syntax

**Definition 1.3** *The set, PreTerm, of pre-terms of Grace over a set $\mathcal{B}$ of term constants, a set $\mathcal{L}$ of labels, and a set $\mathcal{X}$ of term identifiers is given by the following context-free grammar (we assume $x \in \mathcal{X}$, $b \in \mathcal{B}$, $l, m \in \mathcal{L}$ and $\sigma, \tau \in PreType$). The notation $\texttt{opt}(exp)$ indicates that exp is optional in the syntax.*

$$
\begin{array}{lll}
\mathit{codeSequence} & ::= & \mathit{codeUnit}\ \texttt{opt}(\mathit{codeSequence}) \\
\mathit{codeUnit} & ::= & \mathit{declaration} \mid \mathit{statement} \\
\mathit{declaration} & ::= & \mathit{varDeclaration} \mid \mathit{defDeclaration} \mid \mathit{classDeclaration} \mid \\
 & & \mathit{typeDeclaration} \mid \mathit{methodDeclaration} \\
\mathit{statement} & ::= & \mathit{identifier} := \mathit{expression} \mid \texttt{if}(\mathit{expression})\texttt{then}\ \mathit{block}\ \texttt{else}\ \mathit{block} \\
\mathit{innerCodeSequence} & ::= & \mathit{innerCodeUnit}\ \texttt{opt}(\mathit{innerCodeSequence}) \\
\mathit{innerCodeUnit} & ::= & \mathit{innerDeclaration} \mid \mathit{statement} \\
\mathit{innerDeclaration} & ::= & \mathit{varDeclaration} \mid \mathit{defDeclaration} \mid \mathit{classDeclaration} \mid \\
 & & \mathit{typeDeclaration} \\
\mathit{varDeclaration} & ::= & \texttt{var}\ \ \mathit{identifier} : \mathit{typeExpression}\ \ \texttt{opt}(:= \mathit{expression}) \\
\mathit{defDeclaration} & ::= & \texttt{def}\ \mathit{identifier}\!: \mathit{typeExpression} = \mathit{expression} \\
\mathit{methodDeclaration} & ::= & \texttt{method}\ \mathit{identifier}\ \texttt{opt}(\mathit{genericFormals})\ \mathit{methodFormals} \\
 & & \rightarrow \mathit{nonEmptyTypeExpression}\ \texttt{opt}(\mathit{typeConstraints}) \\
 & & \{\mathit{innerCodeSequence}\} \\
\mathit{genericFormals} & ::= & [\![t_1, \ldots, t_n]\!] \\
\mathit{methodFormals} & ::= & (\mathit{id}_1\!: \mathit{typeExp}_1, \ldots, \mathit{id}_n\!: \mathit{typeExp}_n) \\
\mathit{typeConstraints} & ::= & \texttt{where}\ t_1 <:\ \mathit{typeExp}_1, \ldots, t_n <:\ \mathit{typeExp}_n \\
\mathit{typeDeclaration} & ::= & \texttt{type}\ \mathit{typeId}\ \texttt{opt}(\mathit{genericFormals}) = \mathit{typeExpression} \\
\mathit{classDeclaration} & ::= & \texttt{class}\ \mathit{cid.constrId}\ \texttt{opt}(\mathit{genericFormals})\ \texttt{opt}(\mathit{methodFormals}) \\
 & & \rightarrow\ \mathit{typeExp}\ \texttt{opt}(\mathit{typeConstraints}) \\
 & & \{\texttt{opt}(\texttt{inherit}\ \mathit{scid}\langle t_1 \ldots t_n\rangle(e_1, \ldots, e_m)) \\
 & & \mathit{codeSequence}\} \\
\mathit{objectLiteral} & ::= & \texttt{object}\{\texttt{opt}(\texttt{inherit}\ \mathit{scid}\langle t_1 \ldots t_n\rangle(e_1, \ldots, e_m)) \\
 & & \mathit{codeSequence}\}
\end{array}
$$

The only difference between *declaration* and *innerDeclaration* is that *innerDeclarations* may not include method declarations. The distinction is necessary because a method declaration may not directly contain a method declaration (though it can contain an object which itself has method declarations). The type-checking axioms and rules for Grace are given in terms of a type constraint system, $C$, as defined earlier, and an *static type environment*, $E$, which assigns types to free identifiers.

**Definition 1.4** *A* static type environment*, $E$, (with respect to $C$) is a finite set of associations between identifier and type expressions of the form $x\!:\tau$, where each $x$ is unique in $E$ and $C \vdash \tau\!:\texttt{TYPE}$. If the relation $x\!:\tau \in E$, then we write $E(x) = \tau$.*

The collection *Term* of terms of Grace with respect to $C$, $E$ is the set of pre-terms that can be assigned types with respect to the type-assignment axioms and rules in Section 2.

The type-assignment rules provided in Section 2 yield expanded type constraints and assignments as well as types. These expanded type constraints assignments are used to type check the rest of the program. Thus an assertion of the form $C, E \vdash M \diamond \langle \tau, C', E' \rangle$ indicates that if a term $M$ is processed under the type constraint system $C$ and syntactic type assignment $E$, then $M$ has type $\tau$ and the richer syntactic type constraint C' and type assignment $E'$ result. These richer sets will be used in type-checking later terms. If $M$ is a command or declaration then the type $\tau$ will be `Done`.

## 2 Type-checking rules

Typing rules return a triple $\langle \tau, C, E \rangle$ where $\tau$ is the type of the expression and $C$ and $E$ are the new type and variable environments resulting from processing the expression.

For simplicity in the following we ignore the rules for *innerCodeUnit* and *innerCodeSequences* as they duplicate those for *codeUnit* and *codeSequence*. Similarly, we generally assume that optional items are included in syntactic constructs. It is trivial to write the rules without those items.

We will also assume that all "self-inflicted" method invocations (message sends to "self") include an explicit "self" receiver.

### Rules

*Subsumption*
$$\frac{C, E \vdash e \ \diamond \ \langle \tau, C', E' \rangle \quad C \vdash \tau <: \tau'}{C, E \vdash e \ \diamond \ \langle \tau', C', E' \rangle}$$

*CodeSeq*
$$\frac{C, E \vdash \mathit{codeUnit} \ \diamond \ \langle \tau', C', E' \rangle \quad C', E' \vdash \mathit{codeSequence} \ \diamond \ \langle \tau'', C'', E'' \rangle}{C, E \vdash \mathit{codeUnit}\ \mathit{codeSequence} \ \diamond \ \langle \tau'', C'', E'' \rangle}$$

*VarDcl*
$$\frac{C \vdash \tau <: \texttt{Done}}{C, E \vdash \texttt{var}\ x{:}\,\tau \quad \diamond \ \langle \texttt{Done}, C, E\texttt{+}\{x{:}\,\texttt{ref}\ \tau\} \rangle}$$

*VarDclInit*
$$\frac{C, E \vdash \mathit{exp} \diamond \langle \tau, \_, \_ \rangle}{C, E \vdash \texttt{var}\ x{:}\,\tau{:}= \mathit{exp} \quad \diamond \ \langle \texttt{Done}, C, E\texttt{+}\{x{:}\,\texttt{ref}\ \tau\} \rangle}$$

*DefDcl*
$$\frac{C, E \vdash \mathit{exp}{:}\, \langle \tau, \_, \_ \rangle}{C, E \vdash \texttt{def}\ x{:}\,\tau \ = \mathit{exp} \quad \diamond \ \langle \texttt{Done}, C, E\texttt{+}\{x{:}\,\tau\} \rangle}$$

*MethodDcl*
$$\frac{C\texttt{+}\mathit{newConstraints}, E\texttt{+}\mathit{methodFormals} \vdash \mathit{innerCodeSequence} \diamond \ \langle \tau, \_, \_ \rangle}{\begin{array}{l} C, E \vdash \texttt{method}\ \mathit{id}\ \mathit{genericFormals}\ (\mathit{methodFormals}) \to \tau \\ \qquad \mathit{typeConstraints}\ \{\mathit{innerCodeSequence}\} \diamond \langle \texttt{Done}, C, E \rangle \end{array}}$$

In the above, if `genericFormals` is empty, then *newConstraints* is empty. Otherwise if *typeConstraints* $= [t_1 <: T_1, \ldots, t_n <: T_n]$ and U is the set of types in *genericFormals* that do not occur on the left side of a type constraint in *typeConstraints*, then
*newConstraints* $= \{t_1 <: T_1, \ldots, t_n <: T_n\} \cup \{t <: \texttt{Object} \mid t \in U\}$.
We require $t_1, \ldots, t_n$ from `typeConstraints` to all occur in *genericFormals.*

*Note that the method id will be added to C when the class as a whole is type-checked rather than here.*

*TypeDef*
$$\begin{array}{c} C, E \vdash \texttt{type}\ \mathit{typeId}\ \mathit{genericFormals} = \mathit{typeExpression} \diamond \\ \langle \texttt{Done}, C\texttt{+}\{\mathit{typeId} = \texttt{TFunc}(\mathit{genericFormals})\ \mathit{typeExpression}\}, E \rangle \end{array}$$

$$\textit{Object}\quad \frac{\begin{array}{c} C, E' \vdash \textit{codeSequence} \diamond \langle \_, \_, \_\rangle, \\ C, E \vdash \textit{supObj} \diamond \langle \sigma, \_, \_\rangle \end{array}}{\begin{array}{l} C, E \vdash \texttt{object}\ \{\texttt{inherits}\ \textit{supObj} \\ \qquad\qquad \textit{codeSequence}\} \diamond \langle \tau, C, E\rangle \end{array}}$$

where $\sigma' = \{m'_1 \colon \sigma'_1, \ldots, m'_p \colon \sigma'_p\}$ *be the type formed by the declared types of all public and confidential methods of supObj. It is an extension of* $\sigma$.
$\tau = \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}$ *is the subtype of* $\sigma$ *formed by adding the declared types of all new public methods.*
$\tau' = \{m_1 \colon \tau_1, \ldots, m_k \colon \tau_k\}$ *is the subtype of* $\sigma'$ *formed by adding the declared types of all new methods, including confidential and private methods.*
$E' = E + \{\texttt{self} \colon \tau', \texttt{super} \colon \sigma'\}$

Note that $\tau$ and $\tau'$ include the types of the implicit methods generated by the public and confidential defs and variable declarations. If `self` is passed out as a parameter then its type must be restricted to $\tau$. Keyword `super` may only be used as a receiver of method requests.

A class of the form

$$\begin{array}{l} \texttt{class}\ \textit{classId genericFormals methodFormals} \\ \qquad\qquad \to\ \textit{typeExp typeConstraints} \\ \qquad \{\texttt{inherits}\ \textit{supObj} \\ \qquad\ \textit{codeSequence}\} \end{array}$$

can be translated as

$$\begin{array}{l} \texttt{method}\ \textit{classid genericFormals methodFormals} \\ \qquad\qquad \to\ \textit{typeExp typeConstraints} \\ \qquad \texttt{object}\ \{\texttt{inherits}\ \textit{supObj} \\ \qquad\qquad\quad \textit{codeSequence}\} \\ \} \end{array}$$

and type-checked via the translation.

$$\textit{MessageSend}\quad \frac{\begin{array}{c} C, E \vdash \textit{obj} \diamond \langle \{m \colon \langle t_1 <: \tau_1, \ldots, t_n <: \tau_n\rangle \gamma_1 \times \ldots \times \gamma_k \to \tau\}, \_, \_\rangle \\ C \vdash \sigma_i <: \tau_i \textit{ for } 1 \le i \le n \\ C, E \vdash e_j \diamond \langle \gamma_j[t_i \mapsto \sigma_i], \_, \_\rangle \textit{for } 1 \le j \le k \end{array}}{C, E \vdash \textit{obj}.m\langle \sigma_1, \ldots, \sigma_n\rangle(e_1, \ldots, e_k) \diamond \langle \tau[t_i \mapsto \sigma_i], C, E\rangle}$$

$$\textit{NumberLit}\quad C, E \vdash \texttt{numberLiteral} \colon \texttt{Number}$$

$$\textit{StringLit}\quad C, E \vdash \texttt{stringLiteral} \colon \texttt{String}$$

$$\textit{True}\quad C, E \vdash \texttt{true} \colon \texttt{Boolean}$$

*False* $$C, E \vdash \texttt{false}\colon \texttt{Boolean}$$

*Cond*
$$\frac{\begin{array}{c} C, E \vdash \mathit{cond} \diamond \langle \texttt{Boolean}, C', E' \rangle \\ C', E' \vdash \mathit{blk} \diamond \langle \texttt{Block0}[\![\tau]\!], C'', E'' \rangle \\ C'', E'' \vdash \mathit{blk}' \diamond \langle \texttt{Block0}[\![\tau']\!], C''', E''' \rangle \end{array}}{C, E \vdash \texttt{if}\ (\mathit{cond})\ \texttt{then}\ \mathit{blk}\ \texttt{else}\ \mathit{blk}' \diamond \langle \tau | \tau', C''', E''' \rangle}$$

*Identifier* $$C, E \vdash x\colon \tau \qquad if \quad E(x) = \tau$$

*Assn*
$$\frac{C, E \vdash x \diamond \langle \texttt{ref}\ \tau, C', E' \rangle \quad C', E' \vdash M \diamond \langle \tau', C'', E'' \rangle \quad C', E' \vdash \tau' <: \tau}{C, E \vdash x := M \diamond \langle \texttt{Done}, C'', E'' \rangle}$$

*R − Value*
$$\frac{C, E \vdash M \diamond \langle \texttt{ref}\ \tau, C', E' \rangle}{C, E \vdash M \diamond \langle \tau, C', E' \rangle}$$

*Perhaps we should just mark in the abstract syntax when we need the r-value rather than the l-value of a variable.*

Recall that blocks are abbreviations for objects with a single method apply. The derived rule would look like the following.

*Block*
$$\frac{C, E\texttt{+}\{x_1\colon \tau_1, \ldots, x_n\colon \tau_n\} \vdash M\colon \tau}{C, E \vdash \{x_1\colon \tau_1, \ldots, x_n\colon \tau_n \to M\}\colon \{\mathit{apply}\colon \tau_1 \times \ldots \times \tau_n \to \tau\}}$$

*Match*
$$\frac{\begin{array}{c} C, E \vdash v \diamond \langle \tau, C_0, E_0 \rangle \\ C_0, E_0 \vdash \mathit{blk}_1 \diamond \langle \texttt{Block1}[\![\sigma_1, \tau_1]\!], C_1, E_1 \rangle \\ \ldots \\ C_{n-1}, E_{n-1} \vdash \mathit{blk}_n \diamond \langle \texttt{Block1}[\![\sigma_n, \tau_n]\!], C_n, E_n \rangle \\ C \vdash \tau <: \sigma_1 | \ldots | \sigma_n \end{array}}{C, E \vdash \texttt{match}(v)\ \texttt{case}\ \mathit{blk}_1\ \ldots\ \texttt{case}\ \mathit{blk}_n \diamond \langle \tau_1 | ... | \tau_n, C_n, E'_n \rangle}$$

Deal with cases like $\{7 \to "\mathit{hello}"\}$ later.

Not done yet:

- try-catch, should have the type obtained by variant of the try and catch blocks
- module, should be like object, but has imports to be dealt with
- array (should be line-ups)
- generics
- import statement
- return: method with return should return the type of the value returned — last expression.
- dialect