H# Official Documentation

The official specification for H#

Contents

Abstract

This document is the official document for the H-Sharp (H#) programming language. The H stands for hybrid - as this is a multi-paradigm programming language heavily inspired by C# and Scala. This document will only contain the grammar of the language, the operational semantics as well as the type semantics of the language. The operational semantics may be explained with code samples - but obvious uses will not be explained.

This document will also contain the byte-instruction semantics. The official compiler, is written in C# for productivity purposes while the Virtual Machine is implemented using C++. At no point will this document be documenting the internal processes of those applications.

Grammar

The offical H# grammar. Note: $Id \in \textsc{VarEnv}$ and $TypeId \in \textsc{TypeEnv}$. Both refer to the same grammatical definition, defined by the regular expression:

$$(\_|[\texttt{a}-\texttt{Z}])^{+}(\_|[\texttt{0}-\texttt{9}]|[\texttt{a}-\texttt{Z}])^{*}$$

If multiple elements can occur - and it'd be convenient, the element may be suffixed with 'n' to show it may contain n of such elements. $n \in \{0, 1, 2, \dots\}$. The notation $e_0\texttt{,}\dots\texttt{,}e_n$ represents the one to nth element with a specific separator. The full grammar is defined as follows:

$$
\begin{array}{lcl}
\mathit{CompileUnit} & ::= & \mathit{CompileUnitElement} \\
\\
\mathit{CompileUnitElement} & ::= & \mathit{CompileUnitElement}\ \mathit{CompileUnitElement} \\
 & | & \mathit{Directive} \mid \mathit{Scope} \mid \mathit{ScopeElement} \mid \mathit{Declaration} \\
\\
\mathit{Scope} & ::= & \{\ \mathit{ScopeElement}\ \} \\
\\
\mathit{ScopeElement} & ::= & \mathit{ScopeElement}\ \mathit{ScopeElement} \\
 & | & \mathit{Expr}\texttt{;} \mid \mathit{Statement} \mid \mathit{VarDeclaration}\texttt{;} \\
\\
\mathit{Expr} & ::= & \mathit{Expr} \mid \texttt{(}\mathit{Expr}\texttt{)} \mid \mathit{Scope} \mid \mathit{LambdaExpr} \\
 & | & \mathit{Expr}\ \mathit{BinaryOp}\ \mathit{Expr} \mid \mathit{UnaryOp}\ \mathit{Expr} \mid \mathit{Id}\ \mathit{UnaryOp} \\
 & | & \mathit{Expr}\ \texttt{?}\ \mathit{Expr}\ \texttt{:}\ \mathit{Expr} \\
 & | & \mathit{Id} \mid \mathit{Name} \mid \texttt{this} \mid \texttt{base} \\
 & | & \mathit{Expr}\texttt{(}\mathit{Argument}\texttt{)} \mid \mathit{Expr}\texttt{[}\mathit{Expr}\texttt{]} \\
 & | & \texttt{new}\ \mathit{TypeId}\texttt{(}\mathit{Argument}\texttt{)} \mid \texttt{new}\ \mathit{TypeId}\texttt{[}\mathit{Argument}\texttt{]} \\
 & | & \mathit{Expr}\texttt{.}\mathit{Id} \mid \mathit{Expr}\texttt{?.}\mathit{Id} \\
 & | & \mathit{Expr}\texttt{.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)} \mid \mathit{Expr}\texttt{?.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)} \\
 & | & \mathit{Expr}\ \texttt{is}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is null}\ \mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id} \\
 & | & \mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \texttt{is not}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is not null} \\
 & | & \mathit{Expr}\ \texttt{is not}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr} \\
 & | & \mathit{Expr}\ \texttt{as}\ \mathit{TypeId} \\
 & | & \texttt{(}\mathit{TypeID}\texttt{)}\ \mathit{Expr} \\
 & | & \texttt{sizeof(}\mathit{TypeId}\texttt{)} \mid \texttt{addressof(}\mathit{Id}\texttt{)} \mid \texttt{typeof(}\mathit{Expr}\texttt{)} \\
 & | & \mathit{Assignment} \\
 & | & \mathit{Literal} \\
 & | & \mathit{RangeCheck} \\
\\
\mathit{LambdaExpr} & ::= & \texttt{(}\mathit{Param}\texttt{)}\ \texttt{=>}\ \mathit{Expr} \\
\\
\mathit{Directive} & ::= & \texttt{type}\ \mathit{Id}\ \texttt{=}\ \mathit{TypeId}\texttt{;} \\
 & | & \texttt{using}\ \mathit{Name}\texttt{;} \mid \texttt{using}\ \mathit{TypeId}\ \texttt{from}\ \mathit{Name}\texttt{;} \\
 & | & \mathit{NamespaceDecl} \\
\\
\mathit{NamespaceDecl} & ::= & \texttt{namespace}\ \mathit{Name}\ \mathit{Scope} \\
\\
\mathit{Statement} & ::= & \mathit{Assignment}\texttt{;} \mid \mathit{ControlStatement} \mid \mathit{MatchStatement}\texttt{;} \\
 & | & \mathit{TryCatchStatement}
\end{array}
$$

ControlStatement ::= `if` *Expr Scope*
| `if` *Expr Scope* `else` *Scope*
| `if` *Expr Scope* `else if` *Expr Scope*
| `if` *Expr Scope* `else if` *Expr Scope* `else` *Scope*
| `while` *Expr Scope*
| `do` *Scope* `while` *Expr*`;`
| `for (`*Assignment*`;` *Expr*`;` *Expr*`)` *Scope*
| `for (`*VarDeclaration*`;` *Expr*`;` *Expr*`)` *Scope*
| `for (`*TypeId Id* `:` *Expr*`)` *Scope*
| `throw` *TypeID*`(`*Argument*`)`
| `return` *Expr*`;`
| `break;`

MatchStatement ::= *Expr* `match` { *MatchCase* }

MatchCase ::= *MatchCase*`,` *MatchCase*
| `case` *Literal* `=>` *Expr* | `case` *TypeId* `=>` *Expr*
| `case` *TypeId Id* `=>` *Expr* | `case` *TypeId Id* `when` *Expr* `=>` *Expr*
| `case (`*MatchCaseId*`) =>` *Expr*
| `case (`*MatchCaseId*`) when` *Expr* `=>` *Expr*
| `case` *TypeId* `(`*MatchCaseId*`) =>` *Expr*
| `case` *TypeId* `(`*MatchCaseId*`) when` *Expr* `=>` *Expr*
| `case _ =>` *Expr*

MatchCaseId ::= *MatchCaseId*`,` *MatchCaseId*
| *Id* | _

TryCatchStatement ::= `try` *Scope* `catch (`*TypeId Id*`)` *Scope*
| `try` *Scope* `catch (`*TypeId Id*`) when` *Expr Scope*
| `try` *Scope* `catch (`*TypeId Id*`)` *Scope* `finally` *Scope*
| `try` *Scope* `catch (`*TypeId Id*`) when` *Expr Scope* `finally` *Scope*

Assignment ::= *Id* `=` *Expr*
| *Id* `+=` *Expr* | *Id* `-=` *Expr*
| *Id* `*=` *Expr* | *Id* `/=` *Expr*
| *Id* `&=` *Expr* | *Id* `|=` *Expr*
| *Id* `%=` *Expr*

$$
\begin{array}{lll}
\mathit{Declaration} & ::= & \mathit{VarDecl} \mid \mathit{FuncDecl} \\
 & \mid & \mathit{ClassDecl} \mid \mathit{StaticClassDecl} \mid \mathit{TraitDecl} \\
 & \mid & \mathit{UnionDecl} \mid \mathit{EnumDecl} \mid \mathit{StructDecl} \\
 & \mid & \mathit{TraitUniversal} \\
\\
\mathit{ClassDecl} & ::= & \mathit{Modifier}\ \texttt{class}\ \mathit{Id}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{class}\ \mathit{Id}\ \texttt{:}\ \mathit{ParamType}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{class}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{class}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \texttt{:}\ \mathit{ParamType}\ \mathit{ClassBody} \\
\\
\mathit{StructDecl} & ::= & \mathit{Modifier}\ \texttt{struct}\ \mathit{Id}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{struct}\ \mathit{Id}\ \texttt{:}\ \mathit{ParamType}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{struct}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{ClassBody} \\
 & \mid & \mathit{Modifier}\ \texttt{struct}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \texttt{:}\ \mathit{ParamType}\ \mathit{ClassBody} \\
\\
\mathit{StaticClassDecl} & ::= & \texttt{object}\ \mathit{Id}\ \mathit{ClassBody} \\
 & \mid & \mathit{AccessMod}\ \texttt{object}\ \mathit{Id}\ \mathit{ClassBody} \\
 & \mid & \texttt{object}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{ClassBody} \\
 & \mid & \mathit{AccessMod}\ \texttt{object}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{ClassBody} \\
\\
\mathit{TraitDecl} & ::= & \texttt{trait}\ \mathit{Id}\ \mathit{ClassBody} \\
 & \mid & \mathit{AccessMod}\ \texttt{trait}\ \mathit{Id}\ \mathit{ClassBody} \\
\\
\mathit{TraitUniversal} & ::= & \texttt{trait}\ \mathit{TypeId}\ \texttt{for}\ \mathit{TypeId}\ \mathit{Scope} \\
\\
\mathit{ClassBody} & ::= & \texttt{;} \mid \{\ \mathit{ClassMember}\ \} \\
\\
\mathit{ClassMember} & ::= & \mathit{ClassMember}\ \mathit{ClassMember} \\
 & \mid & \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{FuncBody} \mid \mathit{AcessMod}\ \mathit{Id}\texttt{(}\mathit{Param}\texttt{)}\ \mathit{FuncBody} \\
 & \mid & \mathit{VarDecl}\texttt{;} \mid \mathit{AcessMod}\ \mathit{VarDecl}\texttt{;} \\
 & \mid & \mathit{FuncDecl} \mid \mathit{ClassDecl} \mid \mathit{UnionDecl} \mid \mathit{EnumDecl} \\
 & \mid & \texttt{event}\ \mathit{TypeId}\ \mathit{id}\texttt{;} \mid \mathit{AccessMod}\ \texttt{event}\ \mathit{TypeId}\ \mathit{id}\texttt{;}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{VarDecl} & ::= & \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{ValueListInitializer} \mid \mathit{StorageMod}\ \mathit{TypeId}\texttt{[]}\ \mathit{Id}\ \texttt{=}\ \mathit{ValueListInitializer} \\
 & \mid & \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Initializer} \mid \mathit{StorageMod}\ \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Initializer} \\
 & \mid & \mathit{TypeId}\ \mathit{Id} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \mathit{Id} \\
 & \mid & \mathit{TypeId}\texttt{[]}\ \mathit{Id} \mid \mathit{StorageMod}\ \mathit{TypeId}\texttt{[]}\ \mathit{Id} \\
 & \mid & \mathit{LambdaType}\ \mathit{Id}\ \texttt{=}\ \mathit{LambdaExpr} \\
 & \mid & \mathit{TupleDecl} \\
 & & \\
\mathit{TupleDecl} & ::= & \texttt{(}\mathit{ParamType}\texttt{)}\ \mathit{Id}\ \texttt{=}\ \texttt{(}\mathit{Argument}\texttt{)} \\
 & \mid & \texttt{(}\mathit{Param}\texttt{)}\ \texttt{=}\ \texttt{(}\mathit{Argument}\texttt{)} \\
 & \mid & \texttt{(}\mathit{ParamType}\texttt{)[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{(}\mathit{Param}\texttt{)[]}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & & \\
\mathit{FuncDecl} & ::= & \mathit{Id}\texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \mathit{Id}\texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{Id}\ \texttt{=}\ \texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \mathit{Id}\ \texttt{=}\ \texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & \mid & \mathit{AccessMod}\ \texttt{const}\ \mathit{Id}\ \texttt{=}\ \texttt{(}\mathit{FuncParam}\texttt{):}\ \mathit{TypeId}\ \mathit{FuncBody} \\
 & & \\
\mathit{FuncBody} & ::= & \mathit{Scope} \mid \texttt{=>}\ \mathit{Expr} \\
 & & \\
\mathit{FuncParam} & ::= & \mathit{FuncParam}\texttt{,}\ \mathit{FuncParam} \mid \mathit{Param} \\
 & \mid & \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Literal} \\
 & & \\
\mathit{UnionDecl} & ::= & \texttt{union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{static union}\ \mathit{Id}\ \{\ \mathit{UnionMember}\ \} \\
 & & \\
\mathit{UnionMember} & ::= & \mathit{UnionMember}\ \mathit{UnionMember} \\
 & \mid & \mathit{TypeId}\ \mathit{Id}\texttt{;} \\
 & & \\
\mathit{EnumDecl} & ::= & \texttt{enum}\ \mathit{Id}\ \{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{enum}\ \mathit{Id}\ \{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \texttt{enum}\ \mathit{Id}\texttt{(}\mathit{EnumMember}\texttt{)}\{\ \mathit{EnumBodyMember}\ \} \\
 & \mid & \mathit{AccessMod}\ \texttt{enum}\ \mathit{Id}\ \texttt{(}\mathit{EnumMember}\texttt{)}\ \{\ \mathit{EnumBodyMember}\ \} \\
 & & \\
\mathit{EnumBodyMember} & ::= & \mathit{EnumMember} \mid \mathit{FuncDecl} \mid \mathit{FuncDecl}\ \mathit{EnumBodyMember} \\
 & & \\
\mathit{EnumMember} & ::= & \mathit{EnumMember}\texttt{,}\ \mathit{EnumMember} \\
 & \mid & \mathit{Id} \mid \mathit{Id}\ \texttt{=}\ \mathit{LiteralNoNull}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Initializer} & ::= & \mathit{ValueListInitializer} \mid \{\ \mathit{KeyValueListElement}\ \} \\
 & \mid & \{\ \mathit{IdValueListElement}\ \} \\
\mathit{ValueListInitializer} & ::= & \{\ \mathit{ValueListElement}\ \} \\
\mathit{ValueListElement} & ::= & \mathit{Expr} \mid \mathit{ValueListElement}\texttt{,}\ \mathit{ValueListElement} \\
\mathit{IdValueListElement} & :: & \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{IdValueListElement}\texttt{,}\ \mathit{IdValueListElement} \\
\mathit{KeyValueListElement} & :: & \texttt{[}\mathit{Expr}\texttt{]}\ \texttt{=}\ \mathit{Expr} \mid \mathit{KeyValueListElement}\texttt{,}\ \mathit{KeyValueListElement} \\
\mathit{Modifier} & ::= & \mathit{StorageMod} \mid \mathit{AccessMod} \mid \mathit{TypeMod} \mid \mathit{CompilerHintMod} \mid \epsilon \\
 & \mid & \mathit{AccessMod}\ \mathit{StorageMod} \mid \mathit{AccessMod}\ \texttt{abstract variant} \\
 & \mid & \mathit{AccessMod}\ \mathit{CompilerHintMod} \mid \mathit{AccessMod}\ \mathit{StorageMod}\ \mathit{CompilerHintMod} \\
 & \mid & \mathit{AccessMod}\ \mathit{StorageMod}\ \mathit{CompilerHintMod} \mid \mathit{AccessMod}\ \mathit{TypeMod} \\
\mathit{StorageMod} & ::= & \texttt{const} \mid \texttt{static} \mid \texttt{override} \mid \texttt{virtual} \mid \texttt{lazy} \\
\mathit{TypeMod} & ::= & \texttt{variant} \mid \texttt{abstract} \\
\mathit{CompilerHintMod} & ::= & \texttt{inline} \mid \texttt{final} \mid \texttt{constexpr} \\
\mathit{AccessMod} & ::= & \texttt{public} \mid \texttt{private} \mid \texttt{protected} \mid \texttt{internal} \mid \texttt{external}
\end{array}
$$

Param ::= *Param* `,` *Param*
| *TypeId Id* | `const` *TypeId Id*
| *ParamType Id*

ParamType ::= *TypeId* | *ParameterizedType* | *ParamType* `,` *ParamType*

LambdaType ::= `(`*ParamType*`):` *TypeId*
| *TypeId* `:` *TypeId*

ParameterizedType ::= <*TypeId*>

Argument ::= *Expr* | *Expr* `,` *Expr*

Name ::= *Id* | *Name*`.`*Name*

RelativeComparison ::= < | > | <= | >=

RelativeOrEqual ::= *RelativeComparisonOp* | `==` | `!=`

BinaryOp ::= `+` | `-` | `*` | `/` | `%` || | && | | |
| & | << | >> | `=>` | `::` | `??` | `..` |
| *RelativeOrEqual*

UnaryOp ::= `-` | `!` | `#` | `++` | `--` | `*`

RangeCheck ::= *NumericOrId RelativeOrEqual NumericOrId RelativeOrEqual NumericOrId*
| *NumericOrId RelativeOrEqual RangeCheck*

NumericOrId ::= *NumericLit* | *Id*

NumericLit :: *IntLit* | *FloatLit* | *DoubleLit*

LiteralNoNull ::= *NumericLit* | *BoolLit* | *CharLit* | *StringLit*

Literal ::= *LiteralNoNull* | *NullLit*

Digit ::= $x \in \{0, 1, 2, 3, 4, 5, 7, 8, 9\}$

Letter ::= $\{\ \mathrm{x} \mid \mathrm{x} \in$ `UTF-8` $\wedge\ x \notin \mathit{Digit}\ \}$

IntLit ::= Digit^{+}

FloatLit ::= Digit^{+}`.`Digit^{+}`f`

DoubleLit ::= Digit^{+}`.`Digit^{+}

CharLit ::= `'`*Letter*`'` | `'`*Letter*`'`

StringLit ::= ¨$(\mathit{Letter}|\mathit{Digit})^{*}$¨

BoolLit ::= `true` | `false`

NullLit ::= `null`

Operational Semantics

The semantics in this document form a base for the operational rules of the language. These rules will be presented in the form of inference rules and explained code samples and generalisations. Inference rules may omit details not of relevance to a specific operational semantic.

The semantics make the assumption that there are at least four basic environments available when evaluating the language. Implementation wise these may be spread out over more specialized environments.

1. ρ: First environment is the variable environment and will contain all the variables in scope. Elements within this environment may consist of their atomic value v or the reference tuple (ℓ, τ). Where ℓ is a pointer to the value in the σ environment. τ is the associated type of the stored object.
2. κ: Second environment is the *instantiable* type environment. Which is a read-only environment. This may be referred to as the *class* environment - as that is the original purpose. This environment is fully populated before evaluation.
3. ϕ: Third environment contains first-order functions.
4. σ: Fourth environment represents the heap and is where heap-allocated objects reside.

Literal Semantics

The first literal semantics we'll define is the integer and natural number literals. From below no order is defined for which to infer integers to by default. Thus we define it to be `Int32` (as is the case in many other languages). We may also drop the suffixes defined in the rules states below.

$$\textsc{Int8 Literal}\quad \frac{i \in \mathbb{Z} \qquad -2^{7} \leq i < 2^{7} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i \Rightarrow v, \sigma}$$

$$\textsc{Int16 Literal}\quad \frac{i \in \mathbb{Z} \qquad -2^{15} \leq i < 2^{15} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i \Rightarrow v, \sigma}$$

$$\textsc{Int32 Literal}\quad \frac{i \in \mathbb{Z} \qquad -2^{31} \leq i < 2^{31} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i \Rightarrow v, \sigma}$$

$$\textsc{Int64 Literal}\quad \frac{i \in \mathbb{Z} \qquad -2^{63} \leq i < 2^{63} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i\texttt{l} \Rightarrow v, \sigma}$$

$$\textsc{UInt8 Literal}\quad \frac{i \in \mathbb{N} \qquad 0 \leq i < 2^{8} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i \Rightarrow v, \sigma}$$

$$\textsc{UInt16 Literal}\quad \frac{i \in \mathbb{N} \qquad 0 \leq i < 2^{16} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i \Rightarrow v, \sigma}$$

$$\textsc{UInt32 Literal}\quad \frac{i \in \mathbb{N} \qquad 0 \leq i < 2^{32} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i\texttt{u} \Rightarrow v, \sigma}$$

$$\textsc{UInt64 Literal}\quad \frac{i \in \mathbb{N} \qquad 0 \leq i < 2^{64} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash i\texttt{ul} \Rightarrow v, \sigma}$$

Likewise there are two floating-point type literals.

$$\textsc{Real32 Literal}\quad \frac{i \in \mathbb{R} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash r\texttt{f} \Rightarrow v, \sigma}$$

$$\textsc{Real64 Literal}\quad \frac{i \in \mathbb{R} \qquad v = i}{\rho, \kappa, \phi, \sigma \vdash r \Rightarrow v, \sigma}$$

The semantics for boolean literals are rather simple:

$$\textsc{True-Literal}\quad \frac{}{\rho, \kappa, \phi, \sigma \vdash \texttt{true} \Rightarrow \text{true}}$$

$$\textsc{False-Literal}\quad \frac{}{\rho, \kappa, \phi, \sigma \vdash \texttt{false} \Rightarrow \text{false}}$$

Lastly, the two character based literals:

$$\textsc{Char-Literal}\quad \frac{c \in \mathit{UTF\text{-}8} \qquad v = c}{\rho, \kappa, \phi, \sigma \vdash \texttt{'}c\texttt{'} \Rightarrow v}$$

$$\textsc{String-Literal}\quad \frac{c_i \in \mathit{UTF\text{-}8} \text{ for } i = 0, \ldots, n \qquad v = c_0 \cdot c_1 \cdot \ldots \cdot c_n}{\rho, \kappa, \phi, \sigma \vdash \texttt{"}c_0, \ldots, c_n\texttt{"} \Rightarrow v}$$

To note, the $\cdot$ operation described in the inference rule above is the concatenation operator.

General Semantics

Some of the more general operations are defined here. That is, operations not reliant on custom-defined objects and their overriding operator behaviour.

Scope and Variable Operations

The first semantics to visit are concerning the very basics of evaluating scopes and variables.

$$\textsc{Atomic-Lookup}\quad \frac{\rho(x) = v \neq (\ell, \tau)}{\rho, \kappa, \phi, \sigma \vdash x \Rightarrow v}$$

$$\textsc{Reference-Lookup}\quad \frac{\rho(x) = (\ell, \tau) \qquad v = \sigma(\ell)}{\rho, \kappa, \phi, \sigma \vdash x \Rightarrow v}$$

Numeric Operations

$$\textsc{Numeric-Addition}\quad \frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \qquad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \qquad v_1, v_2 \in \mathbb{R} \qquad v = v_1 + v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{+}e_2 \Rightarrow v, \sigma''}$$

$$\textsc{Numeric-Subtraction}\quad \frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \qquad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \qquad v_1, v_2 \in \mathbb{R} \qquad v = v_1 - v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{-}e_2 \Rightarrow v, \sigma''}$$

$$\textsc{Numeric-Multiplication}\quad \frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \qquad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \qquad v_1, v_2 \in \mathbb{R} \qquad v = v_1 * v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{*}e_2 \Rightarrow v, \sigma''}$$

$$\textsc{Numeric-Division}\quad \frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \qquad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \qquad v_1, v_2 \in \mathbb{R} \qquad v = \frac{v_1}{v_2}}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{/}e_2 \Rightarrow v, \sigma''}$$

$$\textsc{Numeric-Negation}\quad \frac{\rho, \kappa, \phi, \sigma \vdash e \Rightarrow v, \sigma' \qquad v \in \mathbb{R} \qquad v' = -v}{\rho, \kappa, \phi, \sigma \vdash \texttt{-}e \Rightarrow v', \sigma'}$$

$$\textsc{Numeric-Increment-Post}\quad \frac{v = \rho(x) \qquad v \in \mathbb{R} \qquad v' = v + 1 \qquad \rho[x \mapsto v']}{\rho, \kappa, \phi, \sigma \vdash x\texttt{++} \Rightarrow v}$$

$$\textsc{Numeric-Increment-Pre}\quad \frac{v = \rho(x) \qquad v \in \mathbb{R} \qquad v' = v + 1 \qquad \rho[x \mapsto v']}{\rho, \kappa, \phi, \sigma \vdash \texttt{++}x \Rightarrow v'}$$

$$\textsc{Numeric-Decrement-Post}\quad \frac{v = \rho(x) \qquad v \in \mathbb{R} \qquad v' = v - 1 \qquad \rho[x \mapsto v']}{\rho, \kappa, \phi, \sigma \vdash x\texttt{--} \Rightarrow v}$$

$$\textsc{Numeric-Decrement-Pre}\quad \frac{v = \rho(x) \qquad v \in \mathbb{R} \qquad v' = v - 1 \qquad \rho[x \mapsto v']}{\rho, \kappa, \phi, \sigma \vdash \texttt{--}x \Rightarrow v'}$$

For the last four semantic rules above, it's noteworthy that the increment and decrement operators may only be used on identifiers.

The next couple of numeric rules are for comparison between two numeric values.

Numeric-LessThan-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 < v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{<}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-LessThan-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 \geq v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{<}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Numeric-GreaterThan-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 > v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{>}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-GreaterThan-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 \leq v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{>}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Numeric-LessThanOrEqual-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_{\leq} v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{<=}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-LessThanOrEqual-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 > v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{<=}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Numeric-GreaterThanOrEqual-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 \geq v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{>=}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-GreaterThanOrEqual-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 < v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{>=}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Numeric-IsEqual-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 = v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{==}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-IsEqual-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 \neq v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{==}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Numeric-IsNotEqual-True

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 \neq v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{!=}e_2 \Rightarrow \texttt{true}, \sigma''}$$

Numeric-IsNotEqual-False

$$\frac{\rho, \kappa, \phi, \sigma \vdash e_1 \Rightarrow v_1, \sigma' \quad \rho, \kappa, \psi, \sigma' \vdash e_2 \Rightarrow v_2, \sigma'' \quad v_1, v_2 \in \mathbb{R} \quad v_1 = v_2}{\rho, \kappa, \phi, \sigma \vdash e_1\texttt{!=}e_2 \Rightarrow \texttt{false}, \sigma''}$$

Lastly, there's the range check operation. Semantically we can define it as:

$$\frac{\begin{array}{c}\textsc{Numeric-LessRangeCheck-True}\\ Op_i \in \{<, <=\} \text{ for } i = 0...n \quad \rho, \kappa, \psi, \sigma^{i-1} \vdash e_i \Rightarrow v_i, \sigma^i \text{ for } i = 0...n \\ v_i \in \mathbb{R} \text{ for } i = 0...n \quad \forall (v_i, Op_i, v_{i+1})\ v_i\ Op_i\ v_{i+1}\end{array}}{\rho, \kappa, \phi, \sigma \vdash e_0\ Op_0\ e_1\ ...\ Op_{n-1}\ e_n \Rightarrow \texttt{true}, \sigma^n}$$

Classes

Objects

Structs

Union

Enum

Traits

Pattern-Matching

Type Semantics

$\theta = \mathit{TypeEnv}$ $\gamma = \mathit{TypeLookupEnv}$ $\eta = \mathit{ReferenceTypeEnv}$	$\theta, \gamma, \eta \vdash \mathit{Expr} : \mathit{Type}$ $\theta, \gamma, \eta \vdash \mathit{Decl} : \theta, \gamma$ $\theta, \gamma, \eta \vdash \mathit{Decl} : \theta, \gamma, \eta$	`int <: float <: double <: INumeric`

$$\texttt{typeof}(t, \gamma, \eta) = \begin{cases} \texttt{Ref}(\gamma(t)) & t \in \gamma, \gamma(t) \in \eta \\ \gamma(t) & t \in \gamma, \gamma(t) \notin \eta \\ t & \text{otherwise}^1 \end{cases}$$

$$\texttt{base}(\tau_1, \tau_2) = \begin{cases} \tau_1 & \text{if } t_2 <: t_1 \\ \tau_2 & \text{if } t_1 <: t_2 \end{cases}$$

Additionally, we note that θ is local to the expression while γ and η are global environments[2]. Additionally, $\eta \subset \gamma$ such that no element in η can be an atomic type and must be a type that is defined during compile-time. Another thing to note is τ consists of the tuple (ϕ, μ). Where ϕ is the set of all fields belonging to the type. Unless it's an atomic type, in which case this will be the empty set. μ is the set of all methods.

T-IntLit
$$\frac{i \in \mathbb{N}}{\theta, \gamma, \eta \vdash i : \texttt{int}}$$

T-Identifier
$$\frac{\tau = \theta(id) \quad id \in \theta}{\theta, \gamma, \eta \vdash id : \tau}$$

T-Addition
$$\frac{\theta, \gamma, \eta \vdash e_1 : \tau_1 \quad \theta, \gamma, \eta \vdash e_2 : \tau_2 \quad \tau' = \texttt{base}(\tau_1, \tau_2) \quad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ + } e_2 : \tau'}$$

T-Subtraction
$$\frac{\theta, \gamma, \eta \vdash e_1 : \tau_1 \quad \theta, \gamma, \eta \vdash e_2 : \tau_2 \quad \tau' = \texttt{base}(\tau_1, \tau_2) \quad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ - } e_2 : \tau'}$$

T-Multiplication
$$\frac{\theta, \gamma, \eta \vdash e_1 : \tau_1 \quad \theta, \gamma, \eta \vdash e_2 : \tau_2 \quad \tau' = \texttt{base}(\tau_1, \tau_2) \quad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ * } e_2 : \tau'}$$

T-Division
$$\frac{\theta, \gamma, \eta \vdash e_1 : \tau_1 \quad \theta, \gamma, \eta \vdash e_2 : \tau_2 \quad \tau' = \texttt{base}(\tau_1, \tau_2) \quad \tau' <: \texttt{INumeric}}{\theta, \gamma, \eta \vdash e_1 \texttt{ / } e_2 : \tau'}$$

T-DeclVar
$$\frac{\theta, \gamma, \eta \vdash e : \tau \quad \theta' = \theta[x \mapsto \tau'] \quad \tau' = \texttt{typeof}(t, \gamma, \eta) \quad \tau <: \tau'}{\theta, \gamma, \eta \vdash t\ x \texttt{ = } e : \theta', \gamma, \eta}$$

T-NewObject
$$\frac{\tau = \texttt{typeof}(t, \gamma, \eta) \quad \theta, \gamma, \eta \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n}{\theta, \gamma, \eta \vdash \texttt{new } t\texttt{(}e_1, \ldots, e_n\texttt{)} : \tau}$$

T-FieldAccess
$$\frac{\theta, \gamma, \eta \vdash e : \tau \quad \tau = (\phi, \mu) \quad \tau' = \phi(id) \quad id \in \phi}{\theta, \gamma, \eta \vdash e.id : \tau'}$$

T-MethodAccess
$$\frac{\theta, \gamma, \eta \vdash e : \tau \quad \tau = (\phi, \mu) \quad \tau' = \mu(id) \quad id \in \mu}{\theta, \gamma, \eta \vdash e.id : \tau'}$$

When inferring the type of a scope - the whole set of control paths must be considered. Additionally, the last expression of a scope is returned to the calling scope.

[1] Atomic type, such as `int`, `bool` or `char`.
[2] With respect to current domain as elements in γ may have local definitions not globally visible.

Compilation Semantics

Bytecode Semantics