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:

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

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

$\mathit{CompileUnit}$	$::=$	$\mathit{CompileUnitElement}$
$\mathit{CompileUnitElement}$	$::=$	$\mathit{CompileUnitElement}\ \mathit{CompileUnitElement}$
	$\mid$	$\mathit{Directive} \mid \mathit{Scope} \mid \mathit{ScopeElement} \mid \mathit{Declaration}$
Scope	$::=$	$\{\ \mathit{ScopeElement}\ \}$
$\mathit{ScopeElement}$	$::=$	$\mathit{ScopeElement}\ \mathit{ScopeElement}$
	$\mid$	$\mathit{Expr}\texttt{;} \mid \mathit{Statement} \mid \mathit{VarDeclaration}\texttt{;}$
Expr	$::=$	$\mathit{Expr} \mid \texttt{(}\mathit{Expr}\texttt{)} \mid \mathit{Scope} \mid \mathit{LambdaExpr}$
	$\mid$	$\mathit{Expr}\ \mathit{BinaryOp}\ \mathit{Expr} \mid \mathit{UnaryOp}\ \mathit{Expr} \mid \mathit{Id}\ \mathit{UnaryOp}$
	$\mid$	$\mathit{Expr}\ \texttt{?}\ \mathit{Expr}\ \texttt{:}\ \mathit{Expr}$
	$\mid$	$\mathit{Id} \mid \mathit{Name} \mid \texttt{this} \mid \texttt{base}$
	$\mid$	$\mathit{Expr}\texttt{(}\mathit{Argument}\texttt{)}$
	$\mid$	$\texttt{new}\ \mathit{TypeId}\texttt{(}\mathit{Argument}\texttt{)} \mid \texttt{new}\ \mathit{TypeId}\texttt{[}\mathit{Argument}\texttt{]}$
	$\mid$	$\mathit{Expr}\texttt{.}\mathit{Id} \mid \mathit{Expr}\texttt{?.}\mathit{Id}$
	$\mid$	$\mathit{Expr}\texttt{.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)} \mid \mathit{Expr}\texttt{?.}\mathit{Id}\texttt{(}\mathit{Argument}\texttt{)}$
	$\mid$	$\mathit{Expr}\ \texttt{is}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is null}\ \mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id}$
	$\mid$	$\mathit{Expr}\ \texttt{is}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr}$
	$\mid$	$\mathit{Expr}\ \texttt{is not}\ \mathit{TypeID} \mid \mathit{Expr}\ \texttt{is not null}$
	$\mid$	$\mathit{Expr}\ \texttt{is not}\ \mathit{TypeID}\ \mathit{Id}\ \texttt{where}\ \mathit{Expr}$
	$\mid$	$\mathit{Expr}\ \texttt{as}\ \mathit{TypeId}$
	$\mid$	$\texttt{(}\mathit{TypeID}\texttt{)}\ \mathit{Expr}$
	$\mid$	$\texttt{sizeof(}\mathit{TypeId}\texttt{)} \mid \texttt{addressof(}\mathit{Id}\texttt{)} \mid \texttt{typeof(}\mathit{Expr}\texttt{)}$
	$\mid$	$\mathit{Assignment}$
	$\mid$	$\mathit{Literal}$
$\mathit{LambdaExpr}$	$::=$	$\texttt{(}\mathit{Param}\texttt{)}\ \texttt{=>}\ \mathit{Expr}$
$\mathit{Directive}$	$::=$	$\texttt{type}\ \mathit{Id}\ \texttt{=}\ \mathit{TypeId}\texttt{;}$
	$\mid$	$\texttt{using}\ \mathit{Name}\texttt{;} \mid \texttt{using}\ \mathit{TypeId}\ \texttt{from}\ \mathit{Name}\texttt{;}$
	$\mid$	$\mathit{NamespaceDecl}$
$\mathit{NamespaceDecl}$	$::=$	$\texttt{namespace}\ \mathit{Name}\ \mathit{Scope}$
$\mathit{Statement}$	$::=$	$\mathit{Assignment}\texttt{;} \mid \mathit{ControlStatement} \mid \mathit{MatchStatement}\texttt{;}$
	$\mid$	$\mathit{TryCatchStatement}$
$\mathit{ControlStatement}$	$::=$	$\texttt{if}\ \mathit{Expr}\ \mathit{Scope}$
	$\mid$	$\texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else}\ \mathit{Scope}$
	$\mid$	$\texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else if}\ \mathit{Expr}\ \mathit{Scope}$
	$\mid$	$\texttt{if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else if}\ \mathit{Expr}\ \mathit{Scope}\ \texttt{else}\ \mathit{Scope}$
	$\mid$	$\texttt{while}\ \mathit{Expr}\ \mathit{Scope}$
	$\mid$	$\texttt{do}\ \mathit{Scope}\ \texttt{while}\ \mathit{Expr}\texttt{;}$
	$\mid$	$\texttt{for (}\mathit{Assignment}\texttt{;}\ \mathit{Expr}\texttt{;}\ \mathit{Expr}\texttt{)}\ \mathit{Scope}$
	$\mid$	$\texttt{for (}\mathit{VarDeclaration}\texttt{;}\ \mathit{Expr}\texttt{;}\ \mathit{Expr}\texttt{)}\ \mathit{Scope}$
	$\mid$	$\texttt{for (}\mathit{TypeId}\ \mathit{Id}\ \texttt{:}\ \mathit{Expr}\texttt{)}\ \mathit{Scope}$
	$\mid$	$\texttt{throw}\ \mathit{TypeID}\texttt{(}\mathit{Argument}\texttt{)}$
	$\mid$	$\texttt{return}\ \mathit{Expr}\texttt{;}$
	$\mid$	$\texttt{break;}$

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*

Declaration ::= *VarDecl* | *FuncDecl*
| *ClassDecl* | *StaticClassDecl* | *TraitDecl*
| *UnionDecl* | *EnumDecl* | *StructDecl*
| *TraitUniversal*

ClassDecl ::= *Modifier* `class` *Id* *ClassBody*
| *Modifier* `class` *Id* `:` *ParamType* *ClassBody*
| *Modifier* `class` *Id*`(`*Param*`)` *ClassBody*
| *Modifier* `class` *Id*`(`*Param*`) :` *ParamType* *ClassBody*

StructDecl ::= *Modifier* `struct` *Id* *ClassBody*
| *Modifier* `struct` *Id* `:` *ParamType* *ClassBody*
| *Modifier* `struct` *Id*`(`*Param*`)` *ClassBody*
| *Modifier* `struct` *Id*`(`*Param*`) :` *ParamType* *ClassBody*

StaticClassDecl ::= `object` *Id* *ClassBody*
| *AccessMod* `object` *Id* *ClassBody*
| `object` *Id*`(`*Param*`)` *ClassBody*
| *AccessMod* `object` *Id*`(`*Param*`)` *ClassBody*

TraitDecl ::= `trait` *Id* *ClassBody*
| *AccessMod* `trait` *Id* *ClassBody*

ClassBody ::= `;` | { *ClassMember* }

ClassMember ::= *ClassMember* *ClassMember*
| *Id*`(`*Param*`)` *FuncBody* | *AcessMod* *Id*`(`*Param*`)` *FuncBody*
| *VarDecl*; | *AcessMod* *VarDecl*;
| *FuncDecl* | *ClassDecl* | *UnionDecl* | *EnumDecl*
| `event` *TypeId* *id*; | *AccessMod* `event` *TypeId* *id*;

VarDecl ::= *TypeId Id* = *Expr* | *StorageMod TypeId Id* = *Expr*
| `var` *Id* = *Expr* | *StorageMod* `var` *Id* = *Expr*
| *TypeId Id* | *StorageMod TypeId Id*
| *LambdaType Id* = *LambdaExpr*
| *TupleDecl*

TupleDecl ::= (*ParamType*) *Id* = (*Argument*)
| (*Param*) = (*Argument*)

FuncDecl ::= *Id*(*FuncParam*): *TypeId FuncBody*
| *AccessMod Id*(*FuncParam*): *TypeId FuncBody*
| *Id* = (*FuncParam*): *TypeId FuncBody*
| *AccessMod Id* = (*FuncParam*): *TypeId FuncBody*
| *AccessMod* `const` *Id* = (*FuncParam*): *TypeId FuncBody*

FuncBody ::= *Scope* | => *Expr*

FuncParam ::= *FuncParam*, *FuncParam* | *Param*
| *TypeId Id* = *Literal*

UnionDecl ::= `union` *Id* { *UnionMember* }
| *AccessMod* `union` *Id* { *UnionMember* }
| *AccessMod* `static union` *Id* { *UnionMember* }

UnionMember ::= *UnionMember UnionMember*
| *TypeId Id*;

EnumDecl ::= `enum` *Id* { *EnumBodyMember* }
| *AccessMod* `enum` *Id* { *EnumBodyMember* }
| `enum` *Id*(*EnumMember*){ *EnumBodyMember* }
| *AccessMod* `enum` *Id* (*EnumMember*) { *EnumBodyMember* }

EnumBodyMember ::= *EnumMember* | *FuncDecl* | *FuncDecl EnumBodyMember*

EnumMember ::= *EnumMember*, *EnumMember*
| *Id* | *Id* = *LiteralNoNull*

Modifier ::= *StorageMod* | *AccessMod* | *AccessMod StorageMod* | ϵ

StorageMod ::= `const` | `constexpr` | `static` | `abstract` | `override` | `virtual` | `final`
| `lazy` | `inline`

AccessMod ::= `public` | `private` | `protected` | `internal` | `external`

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*

$TraitUniversal$ ::= $\texttt{trait}\ TypeId\ \texttt{for}\ TypeId\ Scope$

$BinaryOp$ ::= $\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid < \mid > \mid <= \mid >= \mid \texttt{==} \mid \texttt{!=}$
$\mid \ \| \mid \&\& \mid | \mid \& \mid << \mid >> \mid \texttt{=>} \mid \texttt{::} \mid \texttt{??} \mid \texttt{..}$

$UnaryOp$ ::= $\texttt{-} \mid \texttt{!} \mid \texttt{\#} \mid \texttt{++} \mid \texttt{--} \mid \texttt{*}$

$LiteralNoNull$ ::= $IntLit \mid FloatLit \mid DoubleLit \mid BoolLit \mid CharLit \mid StringLit$

$Literal$ ::= $LiteralNoNull \mid NullLit$

$Letter$ ::= $[\texttt{a-Z}]$

$Digit$ ::= $[\texttt{0-9}]$

$IntLit$ ::= $Digit^+$

$FloatLit$ ::= $Digit^+\texttt{.}Digit^+\texttt{f}$

$DoubleLit$ ::= $Digit^+\texttt{.}Digit^+$

$CharLit$ ::= $\texttt{'}Letter\texttt{'} \mid \texttt{'}\backslash Letter\texttt{'}$

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

$BoolLit$ ::= $\texttt{true} \mid \texttt{false}$

$NullLit$ ::= $\texttt{null}$

Operational Semantics

General Semantics

$$\frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = v \neq (\ell, \omega, \sigma)}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}\ \textsc{VariableLookup}$$

$$\frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{O}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}\ \textsc{HeapObjectLookup}$$

$$\frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{S}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}\ \textsc{HeapStringLookup}$$

$$\frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathtt{A}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}\ \textsc{HeapArrayLookup}$$

Classes

Objects

Structs

Union

Enum

Traits

Type Semantics

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

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

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

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.

$$\textsc{T-IntLit}\quad \frac{i \in \mathbb{N}}{\theta, \gamma, \eta \vdash i : \texttt{int}} \qquad\qquad \textsc{T-Identifier}\quad \frac{\tau = \theta(id) \quad id \in \theta}{\theta, \gamma, \eta \vdash id : \tau}$$

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

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

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

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

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

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

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

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