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: $Id \in$ VarEnv and $TypeId \in$ 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 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:

CompileUnit ::= *CompileUnitElement*

CompileUnitElement ::= *CompileUnitElement CompileUnitElement*
  | *Directive* | *Scope* | *ScopeElement* | *Declaration*

Scope ::= { *ScopeElement* }

ScopeElement ::= *ScopeElement ScopeElement*
  | *Expr*`;` | *Statement* | *VarDeclaration*`;`

Expr ::= *Expr* | `(`*Expr*`)` | *Scope* | *LambdaExpr*
  | *Expr BinaryOp Expr* | *UnaryOp Expr* | *Id UnaryOp*
  | *Expr* `?` *Expr* `:` *Expr*
  | *Id* | *Name* | `this` | `base`
  | *Expr*`(`*Argument*`)`
  | `new` *TypeId*`(`*Argument*`)` | `new` *TypeId*`[`*Argument*`]`
  | *Expr*`.`*Id* | *Expr*`?.`*Id*
  | *Expr*`.`*Id*`(`*Argument*`)` | *Expr*`?.`*Id*`(`*Argument*`)`
  | *Expr* `is` *TypeID* | *Expr* `is null` *Expr* `is` *TypeID Id*
  | *Expr* `is` *TypeID Id* `where` *Expr*
  | *Expr* `is not` *TypeID* | *Expr* `is not null`
  | *Expr* `is not` *TypeID Id* `where` *Expr*
  | *Expr* `as` *TypeId*
  | `(`*TypeID*`)` *Expr*
  | `sizeof(`*TypeId*`)` | `addressof(`*Id*`)` | `typeof(`*Expr*`)`
  | *Assignment*
  | *Literal*

LambdaExpr ::= `(`*Param*`) =>` *Expr*

Directive ::= `type` *Id* `=` *TypeId*`;`
  | `using` *Name*`;` | `using` *TypeId* `from` *Name*`;`
  | *NamespaceDecl*

NamespaceDecl ::= `namespace` *Name Scope*

Statement ::= *Assignment*`;` | *ControlStatement* | *MatchStatement*`;`
  | *TryCatchStatement*

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*
  | `foreach (`*TypeId Id* in *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

Declaration         ::= VarDecl | FuncDecl
                     |  ClassDecl | StaticClassDecl | InterfaceDecl
                     |  UnionDecl | EnumDecl | StructDecl

ClassDecl           ::= Modifier class Id { ClassMember }
                     |  Modifier class Id : TypeId { ClassMember }
                     |  Modifier class Id(Param) { ClassMember } }
                     |  Modifier class Id(Param) : TypeId { ClassMember } }

StructDecl          ::= Modifier struct Id { ClassMember }
                     |  Modifier struct Id : TypeId { ClassMember }
                     |  Modifier struct Id(Param) { ClassMember }
                     |  Modifier struct Id(Param) : TypeId { ClassMember }

StaticClassDecl     ::= object Id { ClassMember }
                     |  AccessMod object Id { ClassMember }
                     |  object Id(Param) { ClassMember }
                     |  AccessMod object Id(Param) { ClassMember }

InterfaceDecl       ::= interface Id { ClassMember }
                     |  AccessMod interface Id { 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;
```

$$
\begin{array}{lll}
\mathit{VarDecl} & ::= & \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \mid \mathit{StorageMod}\ \texttt{var}\ \mathit{Id}\ \texttt{=}\ \mathit{Expr} \\
 & \mid & \mathit{TypeId}\ \mathit{Id} \mid \mathit{StorageMod}\ \mathit{TypeId}\ \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{)} \\
\\
\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} \\
\\
Modifier & ::= & \mathit{StorageMod} \mid \mathit{AccessMod} \mid \mathit{AccessMod}\ \mathit{StorageMod} \mid \epsilon \\
\\
\mathit{StorageMod} & ::= & \texttt{const} \mid \texttt{constexpr} \mid \texttt{static} \mid \texttt{abstract} \mid \texttt{override} \mid \texttt{virtual} \mid \texttt{final} \mid \texttt{lazy} \\
\\
\mathit{AccessMod} & ::= & \texttt{public} \mid \texttt{private} \mid \texttt{protected} \mid \texttt{internal} \mid \texttt{external} \\
\\
\mathit{Param} & ::= & \mathit{Param}\texttt{,}\ \mathit{Param} \\
 & \mid & \mathit{TypeId}\ \mathit{Id} \mid \texttt{const}\ \mathit{TypeId}\ \mathit{Id} \\
 & \mid & \mathit{ParamType}\ \mathit{Id} \\
\\
\mathit{ParamType} & ::= & \mathit{TypeId} \mid \mathit{TypeId}\texttt{,}\ \mathit{TypeId} \\
\\
\mathit{Argument} & ::= & \mathit{Expr} \mid \mathit{Expr}\texttt{,}\ \mathit{Expr} \\
\\
Name & ::= & \mathit{Id} \mid Name\texttt{.}Name
\end{array}
$$

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

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

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

LiteralNoNull ::= *IntLit* | *FloatLit* | *DoubleLit* | *BoolLit* | *CharLit* | *StringLit*

Literal ::= *LiteralNoNull* | *NullLit*

Letter ::= [`a-Z`]

Digit ::= [`0-9`]

IntLit ::= $Digit^+$

FloatLit ::= $Digit^+$`.`$Digit^+$`f`

DoubleLit ::= $Digit^+$`.`$Digit^+$

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

StringLit ::= ¨(*Letter*|*Digit*)*¨

BoolLit ::= `true` | `false`

NullLit ::= `null`

Operational Semantics

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

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

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

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

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.

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

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

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

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

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

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

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

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

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