

# H# Official Documentation

The official specification for H#

## Contents

<b>Abstract</b>	<b>1</b>
<b>Grammar</b>	<b>2</b>
<b>Operational Semantics</b>	<b>6</b>
<b>Type Semantics</b>	<b>7</b>
<b>Compilation Semantics</b>	<b>8</b>
<b>Bytecode Semantics</b>	<b>9</b>

## **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 official H# grammar. Note:  $Id \in \text{VARENV}$  and  $TypeId \in \text{TYPEENV}$ . Both refer to the same grammatical definition, defined by the regular expression:

$$(\_|[a-z])^+(\_|[0-9]|[a-z])^*$$

If multiple elements can occur - and it'd be convenient, the element be suffixed 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* ::= **class** *Id* { *ClassMember* }  
 | *AccessMod* **class** *Id* { *ClassMember* }  
 | *StorageMod* **class** *Id* { *ClassMember* }  
 | *AccessMod* *StorageMod* **class** *Id* { *ClassMember* }  
 | **class** *Id*(*Param*) { *ClassMember* }  
 | *AccessMod* **class** *Id*(*Param*) { *ClassMember* }  
 | *StorageMod* **class** *Id*(*Param*) { *ClassMember* }  
 | *AccessMod* *StorageMod* **class** *Id*(*Param*) { *ClassMember* }

*StructDecl* ::= **struct** *Id* { *ClassMember* }  
 | *AccessMod* **struct** *Id* { *ClassMember* }  
 | *StorageMod* **struct** *Id* { *ClassMember* }  
 | *AccessMod* *StorageMod* **struct** *Id* { *ClassMember* }  
 | **struct** *Id*(*Param*) { *ClassMember* }  
 | *AccessMod* **struct** *Id*(*Param*) { *ClassMember* }  
 | *StorageMod* **struct** *Id*(*Param*) { *ClassMember* }  
 | *AccessMod* *StorageMod* **struct** *Id*(*Param*) { *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*;

<i>VarDecl</i>	$::= \text{TypeId } Id = \text{Expr} \mid \text{StorageMod } \text{TypeId } Id = \text{Expr}$ $\mid \text{var } Id = \text{Expr} \mid \text{StorageMod } \text{var } Id = \text{Expr}$ $\mid \text{TypeId } Id \mid \text{StorageMod } \text{TypeId } Id$ $\mid \text{LambdaType } Id = \text{LambdaExpr}$ $\mid \text{TupleDecl}$
<i>TupleDecl</i>	$::= (\text{ParamType}) \text{ Id} = (\text{Argument})$ $\mid (\text{Param}) = (\text{Argument})$
<i>FuncDecl</i>	$::= \text{Id}(\text{FuncParam}) : \text{TypeId } \text{FuncBody}$ $\mid \text{AccessMod } \text{Id}(\text{FuncParam}) : \text{TypeId } \text{FuncBody}$ $\mid \text{Id} = (\text{FuncParam}) : \text{TypeId } \text{FuncBody}$ $\mid \text{AccessMod } \text{Id} = (\text{FuncParam}) : \text{TypeId } \text{FuncBody}$ $\mid \text{AccessMod } \text{const } \text{Id} = (\text{FuncParam}) : \text{TypeId } \text{FuncBody}$
<i>FuncBody</i>	$::= \text{Scope} \mid \Rightarrow \text{Expr}$
<i>FuncParam</i>	$::= \text{FuncParam}, \text{FuncParam} \mid \text{Param}$ $\mid \text{TypeId } Id = \text{Literal}$
<i>UnionDecl</i>	$::= \text{union } Id \{ \text{UnionMember} \}$ $\mid \text{AccessMod } \text{union } Id \{ \text{UnionMember} \}$ $\mid \text{AccessMod } \text{static } \text{union } Id \{ \text{UnionMember} \}$
<i>UnionMember</i>	$::= \text{UnionMember } \text{UnionMember}$ $\mid \text{TypeId } Id;$
<i>EnumDecl</i>	$::= \text{enum } Id \{ \text{EnumBodyMember} \}$ $\mid \text{AccessMod } \text{enum } Id \{ \text{EnumBodyMember} \}$ $\mid \text{enum } Id(\text{EnumMember}) \{ \text{EnumBodyMember} \}$ $\mid \text{AccessMod } \text{enum } Id (\text{EnumMember}) \{ \text{EnumBodyMember} \}$
<i>EnumBodyMember</i>	$::= \text{EnumMember} \mid \text{FuncDecl} \mid \text{FuncDecl } \text{EnumBodyMember}$
<i>EnumMember</i>	$::= \text{EnumMember}, \text{EnumMember}$ $\mid Id \mid Id = \text{LiteralNotNull}$
<i>StorageMod</i>	$::= \text{const} \mid \text{constexpr} \mid \text{static} \mid \text{abstract} \mid \text{override} \mid \text{virtual} \mid \text{final} \mid \text{lazy}$
<i>AccessMod</i>	$::= \text{public} \mid \text{private} \mid \text{protected} \mid \text{internal} \mid \text{external}$
<i>Param</i>	$::= \text{Param}, \text{Param}$ $\mid \text{TypeId } Id \mid \text{const } \text{TypeId } Id$ $\mid \text{ParamType } Id$
<i>ParamType</i>	$::= \text{TypeId} \mid \text{TypeId}, \text{TypeId}$
<i>Argument</i>	$::= \text{Expr} \mid \text{Expr}, \text{Expr}$
<i>Name</i>	$::= Id \mid \text{Name}.\text{Name}$

<i>LambdaType</i>	::= ( <i>ParamType</i> ) : <i>TypeId</i>   <i>TypeId</i> : <i>TypeId</i>
<i>BinaryOp</i>	::= +   -   *   /   %   <   >   <=   >=   ==   !=        &&        &   <<   >>   =>   ::   ??   ..
<i>UnaryOp</i>	::= -   !   #   ++   --
<i>LiteralNotNull</i>	::= <i>IntLit</i>   <i>FloatLit</i>   <i>DoubleLit</i>   <i>BoolLit</i>   <i>CharLit</i>   <i>StringLit</i>
<i>Literal</i>	::= <i>LiteralNotNull</i>   <i>NullLit</i>
<i>Letter</i>	::= [a-Z]
<i>Digit</i>	::= [0-9]
<i>IntLit</i>	::= <i>Digit</i> <sup>+</sup>
<i>FloatLit</i>	::= <i>Digit</i> <sup>+</sup> . <i>Digit</i> <sup>+</sup> <i>f</i>
<i>DoubleLit</i>	::= <i>Digit</i> <sup>+</sup> . <i>Digit</i> <sup>+</sup>
<i>CharLit</i>	::= 'Letter'   '\Letter'
<i>StringLit</i>	::= ``( <i>Letter</i>   <i>Digit</i> )*``
<i>BoolLit</i>	::= <b>true</b>   <b>false</b>
<i>NullLit</i>	::= <b>null</b>

# Operational Semantics

VARIABLELOOKUP

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

HEAPOBJECTLOOKUP

$$\frac{\rho, \mu, \phi, \kappa, \sigma \vdash \rho(x) = (\ell, \omega, \sigma) \quad \sigma(\ell) = v \quad \omega = \mathbf{0}}{\rho, \mu, \phi, \kappa, \sigma \vdash x \Rightarrow v, \sigma}$$

HEAPSTRINGLOOKUP

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

HEAPARRAYLOOKUP

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

## Type Semantics

$\theta = \text{TypeEnv}$ $\gamma = \text{TypeLookupEnv}$ $\eta = \text{ReferenceTypeEnv}$	$\theta, \gamma, \eta \vdash \text{Expr} : \text{Type}$ $\theta, \gamma, \eta \vdash \text{Decl} : \theta, \gamma$ $\theta, \gamma, \eta \vdash \text{Decl} : \theta, \gamma, \eta$	<code>int &lt;: float &lt;: double &lt;: INumeric</code>
--	---	--

$\frac{\text{T-INTLIT} \quad i \in \mathbb{N}}{\theta, \gamma, \eta \vdash i : \text{int}}$	$\frac{\text{T-ADDITION} \quad \theta, \gamma, \eta \vdash e_1 : \tau_1 \quad \theta, \gamma, \eta \vdash e_2 : \tau_2 \quad \tau <: \text{INumeric}}{\theta, \gamma, \eta \vdash e_1 + e_2 : \tau}$
$\frac{\text{T-DECLVAR-NONREF} \quad \theta, \gamma, \eta \vdash e : \tau \quad \tau \notin \eta \quad \theta' = \theta[x \mapsto \tau] \quad \tau' = \text{typeof}(t, \gamma) \quad \tau <: \tau'}{\theta, \gamma, \eta \vdash t \ x = e : \theta', \gamma, \eta}$	
$\frac{\text{T-DECLVAR-REF} \quad \theta, \gamma, \eta \vdash e : \tau \quad \tau \in \eta \quad \theta' = \theta[x \mapsto \text{Ref}(\tau')] \quad \tau' = \text{typeof}(t, \gamma) \quad \tau <: \tau'}{\theta, \gamma, \eta \vdash t \ x = e : \theta', \gamma, \eta}$	



## Compilation Semantics

## Bytecode Semantics