

H# Official Documentation

The official specification for H#

Contents

Abstract	1
Grammar	2
Operational Semantics	6
General Semantics	6
Classes	6
Objects	6
Structs	6
Union	6
Enum	6
Traits	6
Type Semantics	7
Compilation Semantics	8
Bytecode Semantics	9

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
| 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*

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* | *AccessMod* *Id*(*Param*) *FuncBody*
 | *VarDecl*; | *AccessMod* *VarDecl*;
 | *FuncDecl* | *ClassDecl* | *UnionDecl* | *EnumDecl*
 | **event** *TypeId* *id*; | *AccessMod* **event** *TypeId* *id*;

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

<i>Param</i>	<code>::= Param, Param</code> <code> typeId Id const typeId Id</code> <code> ParamType Id</code>
<i>ParamType</i>	<code>::= typeId ParameterizedType ParamType, ParamType</code>
<i>LambdaType</i>	<code>::= (ParamType): typeId</code> <code> typeId : typeId</code>
<i>ParameterizedType</i>	<code>::= <typeId></code>
<i>Argument</i>	<code>::= Expr Expr, Expr</code>
<i>Name</i>	<code>::= Id Name.Name</code>
<i>TraitUniversal</i>	<code>::= trait typeId for typeId Scope</code>
<i>BinaryOp</i>	<code>::= + - * / % < > <= >= == !=</code> <code> && & << >> => :: ?? ..</code>
<i>UnaryOp</i>	<code>::= - ! # ++ -- *</code>
<i>LiteralNotNull</i>	<code>::= IntLit FloatLit DoubleLit BoolLit CharLit StringLit</code>
<i>Literal</i>	<code>::= LiteralNotNull NullLit</code>
<i>Letter</i>	<code>::= [a-Z]</code>
<i>Digit</i>	<code>::= [0-9]</code>
<i>IntLit</i>	<code>::= Digit⁺</code>
<i>FloatLit</i>	<code>::= Digit⁺.Digit⁺f</code>
<i>DoubleLit</i>	<code>::= Digit⁺.Digit⁺</code>
<i>CharLit</i>	<code>::= 'Letter' '\Letter'</code>
<i>StringLit</i>	<code>::= ``(Letter Digit)*``</code>
<i>BoolLit</i>	<code>::= true false</code>
<i>NullLit</i>	<code>::= null</code>

Operational Semantics

General Semantics

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

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

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

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

Classes

Objects

Structs

Union

Enum

Traits

Type Semantics

$\theta = TypeEnv$ $\gamma = TypeLookupEnv$ $\eta = ReferenceTypeEnv$	$\theta, \gamma, \eta \vdash Expr : Type$ $\theta, \gamma, \eta \vdash Decl : \theta, \gamma$ $\theta, \gamma, \eta \vdash Decl : \theta, \gamma, \eta$	<code>int <: float <: double <: INumeric</code>
$\text{typeof}(t, \gamma, \eta) = \begin{cases} \text{Ref}(\gamma(t)) & t \in \gamma, \gamma(t) \in \eta \\ \gamma(t) & t \in \gamma, \gamma(t) \notin \eta \\ t & \text{otherwise}^1 \end{cases}$		$\text{base}(\tau_1, \tau_2) = \begin{cases} \tau_1 & t_2 <: t_1 \\ \tau_2 & t_1 <: t_2 \end{cases}$

Additionally, we note that θ is local to the expression while γ and η are global environments². 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.

$$\frac{\text{T-INTLIT} \quad i \in \mathbb{N}}{\theta, \gamma, \eta \vdash i : \text{int}}$$

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

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

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

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

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

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

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

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

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

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.

¹Atomic type, such as `int`, `bool` or `char`.

²With respect to current domain as elements in γ may have local definitions not globally visible.

Compilation Semantics

Bytecode Semantics