

H# Official Documentation

The official specification for H#

Contents

Abstract	1
Grammar	2
Operational Semantics	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)
| 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
| Assignment
| Literal

LambdaExpr ::= (Param) => Expr

Directive ::= type Id = TypeId;
| using Id; | using TypeId from Id;

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** *Literal* => *Scope*
 | **case** *_* => *Expr* | **case** *_* => *Scope*

TryCatchStatement ::= **try** *Scope* **catch** *TypeId* *Id* *Scope*
 | **try** *Scope* **catch** *TypeId* *Id* *Scope* **finally** *Scope*

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

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

NamespaceDecl ::= **namespace** *Name* *Scope*

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*
 | *VarDecl*; | *AccessMod* *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*
| *LambdaType* *Id* => *LambdaExpr*

FuncDecl ::= *Id*(*Param*): *TypeId* *Scope*
| *AccessMod* *Id*(*Param*): *TypeId* *Scope*
| *Id* = (*Param*): *TypeId* *Scope*
| *AccessMod* *Id* = (*Param*): *TypeId* *Scope*
| *AccessMod* **const** *Id* = (*Param*): *TypeId* *Scope*

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*

StorageMod ::= **const** | **static** | **abstract** | **override** | **virtual** | **final** | **lazy**

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

Param ::= *Param*, *Param*
| *TypeId* *Id* | **const** *TypeId* *Id*
| *LambdaType* *Id*

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

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

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

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

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

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

Compilation Semantics

Bytecode Semantics