

Hi - a tiny language

It illustrates in minimal form:

1. An infinite type system.
2. Functions and parameter passing.
3. Computationally complete control structures.
4. Operator overloading.
5. Separate compilation.
6. The possibility of parallel evaluation of code.

Hi has a single base type: the scalar. These may be treated as either ASCII characters or as numbers. The representation of numbers - the number of bits used or whether reals or integers are used is up to the implementor. I will present an implementation using 32 bit integers, you may be asked to use a different implementation.

In the source code of the program words can be represented as decimal numbers. Thus 7, 42, 99 are valid literal representations of scalars. Minus numbers are written thus:

`_50`

using the underbar character. This is done to make parsing easier by ensuring that - is always treated as a binary operator.

The type of a scalar is named `scalar`.

Vectors

Vectors of arbitrary dimension are supported. A one dimensional vector constructor can be written thus: [67, 90, 12]

or thus 'here we go'

In the latter case any sequence of ASCII characters can be embedded in the string. The Pascal sort of escape mechanism is used, thus "" is the representation of the single quote character.

An n dimensional vector constructor for $n > 1$ is written as a comma separated list of $n - 1$ dimensional vector values in [].

Vector type names

The type of an n dimensional vector is named $vecn$, thus: $vec1$, $vec3$ are valid vector type names. The name $vec0$ is an alternative name for the type name $scalar$. The number which follows the word vec in a type is called the *rank* of the type.

Null vectors

A null vector is written either as $:\ []$ or $''$. A null vector is of type $vec1$.

Functions

A function maps a list of arguments to a result. Function calls have the form: `foo(p,q,r)` Where `foo` is a function name and `p,q,r` are values.

An example function declaration would be

```
dbl(scalar a->scalar) a+a
```

In this `dbl` is the name of the function, `(scalar -> scalar)` is the type of the function and `a+a` is the expression that evaluates to give the function result. The identifier `a` names the parameter to the function.. Parameters must be typed.

Parameter names are sequences of alphabetic characters.

Function names are either sequences of alphabetic characters or, sequences of one or more characters drawn from the operator alphabet

+ - = \ / . ! # & @ * ~ | % _ ? < > :

A sequence drawn from the operator alphabet is an *operator name*.

A function declared with an operator name must have two parameters.

External functions (written in C for instance) can be denoted using the word `external` in place of the expression that is evaluated to give a result.

Expressions

An expression is a formula that is evaluated to yield a result. Expressions are either:

1. primary values, which are:
 - (a) literal values (section)
 - (b) parameter names (section)
 - (c) vector constructors (section)
 - (d) conditional values
 - (e) function calls
 - (f) bracketed expressions

2. operator expressions

Primary values

Function calls

A function call has the syntactic form $f(x, y, , , , z)$ where f is a function name, and $x, y, ..etc$ are values. The types of the parameter names declared in the function definition must exactly match the types of the corresponding values supplied as actual parameters except as described in the section on operator overloading .

Bracketed expressions

Any expression p can be enclosed in brackets thus (p) . Bracketing imposes an association order on operator expressions.

Conditional Evaluation

An expression can be conditionally evaluated using the `if .. then ... else ... fi` thus where `x=4`

```
if x>2 then a else b fi
```

will evaluate to `a`. The conditional expression evaluates to the `then` branch if the expression between `if` and `then` is non zero. Otherwise it evaluates to the `else` branch.

Operator expressions

In Hi all operators are of the same priority. Expressions are evaluated right to left. The predefined operators and their signatures are:

Operator	Signature	
+	(scalar , scalar \rightarrow scalar)	signed addition
*	(scalar , scalar \rightarrow scalar)	integer multiplication
/	(scalar , scalar \rightarrow scalar)	integer division
-	(scalar , scalar \rightarrow scalar)	signed subtraction
<	(scalar , scalar \rightarrow scalar)	less than
>	(scalar , scalar \rightarrow scalar)	greater than
&&	(scalar , scalar \rightarrow scalar)	and
	(scalar , scalar \rightarrow scalar)	or
==	(scalar , scalar \rightarrow scalar)	equal to
#	(vec n , scalar \rightarrow vec $n - 1$)	column subselect
##	(vec2, scalar \rightarrow vec1)	matrix row select
	(vec n , vec n \rightarrow vec n)	concatenate
~	(scalar \rightarrow scalar)	not, maps 0-1
-	(scalar \rightarrow scalar)	negate, multiply by -1

Examples of operator expressions are:

<code>2+2</code>	<code>= 4</code>
<code>a-b*c</code>	<code>evaluates as a-(b*c)</code>
<code>1-2*3</code>	<code>= _5</code>
<code>a*b-c</code>	<code>evaluates as a*(b-c)</code>
<code>2*3-1</code>	<code>= 4</code>
<code>[1,2,3] [7,11,13]</code>	<code>= [1,2,3,7,11,13]</code>
<code>[[11,12],[21,22]]#2</code>	<code>= [12,22]</code>
<code>[[11,12],[21,22]]##2</code>	<code>= [21,22]</code>
<code>[1,2]#1</code>	<code>= 1</code>
<code>'hi' 'lo'</code>	<code>= 'hilo'</code>

Subscripted vectors

Examples

<code>[1,2,3,5,7]#(4)</code>	<code>= 5</code>
<code>[1,2,3,5,7]#([3,2,4])</code>	<code>= [3,2,5]</code>
<code>'hello'#(3)</code>	<code>= 'l'</code>
<code>'hello'#[2,3,5,1]</code>	<code>= 'eloh'</code>

`['me', 'too']##(2) = 'too'`

`[[9], [8,7]]## 2 = [8,7]`

`(['me', 'too']## 2)# [2,1] = 'ot'`

`['me', 'too']#([[1,2], [2,1]]) = [['me', 'ot']]`

`'zot'#([[1,2], [2,1]]) = ['zo', 'oz']`

If a vector has length n then attempting to subscript by numbers outside the closed interval $[1..n]$ will yield undefined results.

Overloading of operators

All primitive and user defined operators are overloaded to work on vectors of higher dimension than those for which they are defined. Examples using predefined operators are:

$2 + [9, 11, 7]$	$= [11, 13, 9]$
$[2, 3] * [3, 4]$	$= [6, 12]$
$[1, 2, 3] + [4, 5]$	$= [5, 7]$
$[1, 2] * [2, 3, 2]$	$= [2, 6]$
$[[1, 2], [3, 4]] - [1, 0]$	$= [[0.2], [2.4]]$

Similarly functions are overloaded so that given

`foo(scalar a, vec1 b, vec2 c -> vec1) a+b*c(a)`

the expression

`foo(1, [[3], [4, 5]], [[10, 20], [30, 40]])`

will yield a result of type `vec2 : [[31], [41, 101]]`.

Primitive functions

The following functions have to be provided by a runtime system and can not be written in Hi itself.

length

The `length(vec1 → scalar)` function returns the number of elements in a vector.

getChar

The function `getChar(→ scalar)` returns the next character from standard input.

putChar

The function `putChar(scalar→ scalar)` outputs a scalar to standard output as a character. It returns its input parameter.

iota

The function `iota(scalar →vec1)` generates a vector containing an ascending sequence of integers. Thus `iota(4) → [1, 2, 3, 4]`

The following functions should be provided in a Hi implementation either in a library written in Hi itself or as externals.

`getNum`

The function `getInt(→ scalar)` parses the input to read the next decimal number.

`putNum`

The function `putInt(scalar→ scalar)` outputs a scalar to standard output as a decimal number. It returns its input parameter.

HiMain function

A Hi program is a sequence of function declarations. One of these must be called `HiMain`. This should be of signature `(\rightarrow scalar)`. This function will be executed as the main program. It should return 0 to indicate success or return a non-zero value to indicate an error code.