Honu

Version 4.2.2

October 4, 2009

Honu is a family of languages built on top of Scheme. Honu syntax resembles Java. Like Scheme, however, Honu has no fixed syntax, because Honu supports extensibility through macros and a base syntax of H-expressions, which are analogous to S-expressions.

The Honu language currently exists only as a undocumented prototype. Scheme's parsing and printing of H-expressions is independent of the Honu language, however, so it is documented here.

Contents

1 H-expressions

The Scheme reader incorporates an H-expression reader, and Scheme's printer also supports printing values in Honu syntax. The reader can be put into H-expression mode either by including `#hx` in the input stream, or by calling `read-honu` or `read-honu-syntax` instead of `read` or `read-syntax`. Similarly, `print` (or, more precisely, the default print handler) produces Honu output when the `print-honu` parameter is set to `#t`.

When the reader encounters `#hx`, it reads a single H-expression, and it produces an S-expression that encodes the H-expression. Except for atomic H-expressions, evaluating this S-expression as Scheme is unlikely to succeed. In other words, H-expressions are not intended as a replacement for S-expressions to represent Scheme code.

Honu syntax is normally used via `#lang honu`, which reads H-expressions repeatedly until an end-of-file is encountered, and processes the result as a module in the Honu language.

Ignoring whitespace, an H-expression is either

- a number (see §1.1 "Numbers");
- an identifier (see §1.2 "Identifiers");
- a string (see §1.3 "Strings");
- a character (see §1.4 "Characters");
- a sequence of H-expressions between parentheses (see §1.5 "Parentheses, Brackets, and Braces");
- a sequence of H-expressions between square brackets (see §1.5 "Parentheses, Brackets, and Braces");
- a sequence of H-expressions between curly braces (see §1.5 "Parentheses, Brackets, and Braces");
- a comment followed by an H-expression (see §1.6 "Comments");
- `#;` followed by two H-expressions (see §1.6 "Comments");
- `#hx` followed by an H-expression;
- `#sx` followed by an S-expression (see §12.6 "The Reader").

Within a sequence of H-expressions, a sub-sequence between angle brackets is represented specially (see §1.5 "Parentheses, Brackets, and Braces").

Whitespace for H-expressions is as in Scheme: any character for which `char-whitespace?` returns true counts as a whitespace.

1.1 Numbers

The syntax for Honu numbers is the same as for Java. The S-expression encoding of a particular H-expression number is the obvious Scheme number.

1.2 Identifiers

The syntax for Honu identifiers is the union of Java identifiers plus `;`, `,`, and a set of operator identifiers. An *operator identifier* is any combination of the following characters:

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

The S-expression encoding of an H-expression identifier is the obvious Scheme symbol.

Input is parsed to form maximally long identifiers. For example, the input `int->int;` is parsed as four H-expressions represented by symbols: `'int`, `'->`, `'int`, and `'|;|`.

1.3 Strings

The syntax for an H-expression string is exactly the same as for an S-expression string, and an H-expression string is represented by the obvious Scheme string.

1.4 Characters

The syntax for an H-expression character is the same as for an H-expression string that has a single content character, except that a `'` surrounds the character instead of `"`. The S-expression representation of an H-expression character is the obvious Scheme character.

1.5 Parentheses, Brackets, and Braces

A H-expression between `(` and `)`, `[` and `]`, or `{` and `}` is represented by a Scheme list. The first element of the list is `'#%parens` for a `(...)` sequence, `'#%brackets` for a `[...]` sequence, or `'#%braces` for a `{...}` sequence. The remaining elements are the Scheme representations for the grouped H-expressions in order.

In an H-expression sequence, when a `<` is followed by a `>`, and when nothing between the `<` and `>` is an immediate symbol containing a `=`, `&`, or `|`, then the sub-sequence is represented by a Scheme list that starts with `'#%angles` and continues with the elements of the sub-sequence between the `<` and `>` (exclusive). This representation is applied recursively, so that angle brackets can be nested.

An angle-bracketed sequence by itself is not a single H-expression, since the `<` by itself is a single H-expression; the angle-bracket conversion is performed only when representing sequences of H-expressions.

Symbols with a `=`, `&`, or `|` prevent angle-bracket formation because they correspond to operators that normally have lower or equal precedence compared to less-than and greater-than operators.

1.6 Comments

An H-expression comment starts with either `//` or `/*`. In the former case, the comment runs until a linefeed or return. In the second case, the comment runs until `*/`, but `/*`...`*/` comments can be nested. Comments are treated like whitespace.

A `#;` starts an H-expression comment, as in S-expressions. It is followed by an H-expression to be treated as whitespace. Note that `#;` is equivalent to `#sx#;#hx`.

1.7 Honu Output Printing

Some Scheme values have a standard H-expression representation. For values with no H-expression representation but with a `read`able S-expression form, the Scheme printer produces an S-expression prefixed with `#sx`. For values with neither an H-expression form nor a `read`able S-expression form, then printer produces output of the form `#<`...`>`, as in Scheme mode. The `print-honu` parameter controls whether Scheme's printer produces Scheme or Honu output.

The values with H-expression forms are as follows:

- Every real number has an H-expression form, although the representation for an exact, non-integer rational number is actually three H-expressions, where the middle H-expression is `/`.
- Every character string is represented the same in H-expression form as its S-expression form.
- Every character is represented like a single-character string, but (1) using a `'` as the delimiter instead of `"`, and (2) protecting a `'` character content with a `\` instead of protecting `"` character content.
- A list is represented with the H-expression sequence `list(`⟨*v*⟩`,` ...`)`, where each ⟨*v*⟩ is the representation of each element of the list.
- A pair that is not a list is represented with the H-expression sequence `cons(`⟨*v1*⟩`,` ⟨*v2*⟩`)`, where ⟨*v1*⟩ and ⟨*v2*⟩ are the representations of the pair elements.

- A vector's representation depends on the value of the `print-vector-length` parameter. If it is `#f`, the vector is represented with the H-expression sequence `vectorN(`⟨*v*⟩`, ...)`, where each ⟨*v*⟩ is the representation of each element of the vector. If `print-vector-length` is set to `#t`, the vector is represented with the H-expression sequence `vectorN(`⟨*n*⟩`, `⟨*v*⟩`, ...)`, where ⟨*n*⟩ is the length of the vector and each ⟨*v*⟩ is the representation of each element of the vector, and multiple instances of the same value at the end of the vector are represented by a single ⟨*v*⟩.
- The empty list is represented as the H-expression `null`.
- True is represented as the H-expression `true`.
- False is represented as the H-expression `false`.