NAME

simp — simplistic programming language

DESCRIPTION

The **simp** programming language is a minimalist (yet expressive), statically scoped, dynamically typed, properly tail-recursive, general-purpose LISP-1 programming language. All manipulable entities in **simp** are first-class objects. Imperative, functional, and message-passing programming styles (to name a few) may be conveniently expressed in the **simp** programming language.

simp (like most dialects of Lisp) employs a fully parenthesized prefix notation for programs and other data known as *S-expression* (or "sexp" for short). However, **simp** implements S-expressions as **vectors** rather than cons cells (aka pairs). In fact, it is the vector, not the pair, the primary mean of data combination. **simp** has no notion of **cons**, **car**, **cdr** or other functions common in regular LISP dialects. In **simp** lists can be implemented as a chain of vectors, in which the last element of each vector is a pointer to the next vector.

A **simp** program is a sequence of expressions, possibly alternated with comments and whitespace. Expressions are parenthesized S-expressions. See the section **FORMAL SYNTAX** for explanation on the composition of a **simp** program.

EVALUATION

Evaluation is the process of computing objects from other objects. An object can evaluate to zero or more objects. An object can even evaluate to itself (a so called *self-evaluating* object). An evaluation can do more than just compute objects, such as mutate an object or perform input/output; such additional processes are called *side effects*.

Given an object *obj* and an environment *env*, the algorithm for evaluating *obj* in *env* is as follows:

1. If *obj* is a symbol, look for a binding to it in *env*.
   - If a binding is found, *obj* evaluates to the value it is bound to.
   - If a binding is not found, an error is signaled.
2. If *obj* is not a vector, *obj* evaluates to itself (it is a self-evaluating object).
3. If *obj* is the empty vector, an error is signalled.
4. If *obj* is a non-empty vector, recursivelly evaluate its first element into *env* following this algorithm. This first element is called the *operator* and the remaining elements, if any, are called *operands*.
   - If the operator evaluates to an applicative operation, recursivelly evaluate the operands into *env*. Then *obj* evaluates to the result of the application of the evaluated operands into the operation.
   - If the operator evaluates to an non-applicative operation, then *obj* evaluates to the result of the application of the operands into the operation.
   - If the operator does not evaluates to an operation at all, an error is signalled.

Signalling an error interrupts the algorithm, even when at a deep level of recursion.

Although objects are usually read and then evaluated, those are separate processes, and either can be performed alone. Reading does not evaluate anything; it just converts an external representation of an object to the object itself. For example, the sequence of characters "`(+ 2 6)`" is an external representation of an object that evaluates to the integer 8; it is not however an external representation of the integer 8. The syntax of **simp** has the property that any sequence of characters that is an expression is also the external representation of some object. That can lead to confusion, since it may not be obvious out of context whether a given sequence of characters is intended to denote data or program, but it is also a source of power, since it facilitates writing programs such as interpreters and compilers that treat programs as data (or vice versa).

This manual uses the rightwards double arrow symbol “⇒” (U+21D2) to separate the external representation of an object from the external representation of the objects it evaluates to. For example "**(+ 2 6) ⇒ 8**" means that the object represented by "**(+ 2 6)**" evaluates to the object represented by "**8**".

This manual also uses the triple bar symbol “≡” (U+2261) to separate the external representations of two objects that evaluate to the same objects. For example "**(+ 2 6) ≡ (+ 4 4)**" means that the object represented by "**(+ 2 6)**" and the object represented by "**(+ 4 4)**" evaluate to the same thing.

There are two means of comparing objects. Two objects are the *same* if they denote they are bound to the same thing or location. Two objects are *equivalent* if the values stored in their locations are the same.

Evaluation is subject to the current environment (see below).

Environment

An identifier may name a location in memory where a value can be stored. Such identifier is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's *value*. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

An environment is a sequence of *frames*. Each frame is a table (possible empty) of *bindings*, which associate variable names with their corresponding location in memory. Frames are structured as a tree: each frame can point to either nothing or to a parent tree. The *value of a variable* with respect to an environment is the value on the location given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the sequence specifies a binding for the variable, then the variable is said to be *unbound* in the environment.

The environment is crucial for the evaluation process, because it determines the context in which an expression should be evaluated. Indeed, one could say that expressions do not, in themselves, have any meaning. Rather, an expression aquires a meaning only with respect to some environment in which it is evaluated. Even the interpretation of an expression as straightforward as **(+ 1 1)** depends on an understanding that one is operating in a context in which **+** is the symbol of addition.

simp is a statically scoped language with block structure. To each place where an identifier is bound in a program there corresponds a *region* of the program text within which the binding is visible. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use. If there is no binding of the identifier whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any. If there is no binding for the identifier, it is said to be *unbound*.

External representations

An important concept in **simp** (and Lisp) is that of the *external representation* of an object as a sequence of characters. There are two forms of external representation: the printed and the read forms.

The *printed external representation* of an object is unique; and all objects have a printed external representation. For example, the printed external representation of the integer 28 is the sequence of characters "**28**", and the printed external representation of a vector consisting of the integers 8 and 13 is the sequence of characters "**(8 13)**". The printed external representation of an object is generated by the **write** procedure.

The *read external representation* of an object is not necessarily unique; and certain objects may have no read external representation at all. The integer 28 has the read external representations "**28**", "**28e**", and "**0x1C**", among others. Closures and port objects, for example, have no read external representation. The read external representation of an object is parsed by the **read** procedure.

In most cases, an object's printed external representation is also a valid read external representation for the object. Objects that have no read external representations have their printed external representation surrounded by "**#<**" and "**>**". For example, the printed external representation of the an output port is something like "**#<output-port 0x0123456789ABCDEF>**".

DATA TYPES

A *data type* can be interpreted as a set of possible objects. Each object belong to at least one type. Data types can overlap, and objects can belong to two or more types. A *primitive data type* is a basic data type that is built into **simp** and from which all other data types are constructed. Primitive data types are *disjoint* and do not overlap (that is, each object belongs to one and only one primitive data type).

simp (like in most Lisps) is a dynamically typed programming language. Types are associated with objects rather than with variables. (Statically typed languages, by contrast, associate types with variables and expressions as well as with values). Object are self-typing; the primitive type of each object is implicit in the object itself.

For each primitive data type (and a few other non-primitve ones), the standard library defines a set of variables bound to objects (constants, predicates, constructors, mutators, and accessors) used to manipulate objects of that data type (see **STANDARD LIBRARY**).

Numbers

[TODO: fixnums, bignums, numerical tower, etc]

Booleans

The boolean data type contains only two distinct unique objects: the true and false objects. These objects have no read external representation, therefore they cannot be created by the **read** procedure. They have, however, the printed external representations "#<true>" and "#<false>".

Boolean objects (or "booleans" for short) can be used to control the evaluation of conditional procedures. The procedures in the **STANDARD LIBRARY** interpret the false boolean object as a logical false, and any other object (including the true boolean object) as a logical true.

A boolean is immutable and self-evaluating.

Symbols

The symbol data type contains objects holding an interned string of characters. Symbol objects have identifiers as external representations. Two symbol objects with the same external representation (either read or printed) are the same object (they denote the same location in memory).

Symbol objects (or "symbols" for short) are used to represent identifiers in programs. The printed external representation of a symbol is called the *name* of the symbol.

A symbol is immutable and evaluates to the value bound to the variable with the same name as the symbol in the current environment.

End-of-file

The end-of-file data type contains a single object, called the end-of-file. The end-of-file object has no read external representation. It has, however, the printed external representation "#<eof>".

The end-of-file object (or "eof" for short) is used to represent the end of a read file or program.

The eof is immutable and self-evaluating.

Port

The port data type contains objects representing input and output devices. A port object has no read external representation. The printed external representation of a port is unique for a port object, but unpredictable.

Port objects (or "ports", for short) can be input ports, used to read bytes from files or bytevectors; or output ports, used to write bytes into files or bytevectors. Ports can be closed. When a port is closed, no further input/output operation is permited on that port. Input/output operation can be buffered, and closing a port flushes the buffer.

A port is immutable and self-evaluating.

Byte

The byte data type (also known as "character" data type) contains a value from 0 to 255 inclusive. The external representation of a byte is a character literal.

Bytevectors

The bytevector data type (also known as "string" data type) contains objects denoting a sequence of zero or more locations in memory, each one holding exactly a byte. Where a *byte* is an exact integer in the range from 0 to 255 inclusive. A bytevector is typically more space-efficient than a vector containing the same values. The external representation of bytevectors is a string literal.

Bytevector objects are homogenous structures whose elements are indexed by integers and whose elements can be randomly accessed in constant time. The *length* of a bytevector is the number of elements that it contains. This number is a non-negative integer that is fixed when the bytevector is created. The *valid indexes* of a bytevector are the exact non-negative integers less than the length of the bytevector, starting at index zero.

Bytevectors are usually used to hold string of characters encoded in UTF-8. For example, "Hello World" and "Eĥoŝanĝo Ĉiuĵaŭde" are two strings of characters encoded in UTF-8 in a bytevector. "\x00\x0A\x05" is a bytevector of length 3 containing, in order, the bytes 0, 10 and 5 (or 0, A, and 5, in their hexadecimal form).

A bytevector can be mutable or immutable, and is self-evaluating.

Vectors

The vector data type contains objects denoting a sequence of zero or more locations in memory, each one holding an object of arbitrary type. A vector object can have several different external representations (see below).

Vector objects (or "vectors" for short) are heterogenous structures whose elements are indexed by integers and whose elements can be randomly accessed in constant time. The *length* of a vector is the number of elements that it contains. This number is a non-negative integer that is fixed when the vector is created. The *valid indexes* of a vector are the exact non-negative integers less than the length of the vector. The first element in a vector is indexed by zero, and the last element is indexed by one less than the length of the vector. A vector can contain any object as its elements, even other vectors.

A vector with zero element is called a *nil*. A vector with one element is called a *box*. A vector with two elements is called a *pair*. A vector with a number n of elements is called a *n-tuple*.

More complex data structures, such as linked lists, hash tables, trees, and records (to name a few), can be implemented in terms of vectors.

STANDARD LIBRARY

The standard library only contains operations, which are listed below, followed by an annotation of its operands and resulting evaluated value in parentheses.

Each annotation contains types of operands and the types of evaluation, separated by a right-pointing arrow. An interrocation point (**?**) after a type indicates zero or one object of that type. An asterisk (∗) after a type indicates zero or more objects of that type. A plus sign (**+**) after a type indicates one or more objects of that type. A exclamation point (!) before a type indicates that the operand is not evaluated. The empty symbol (∅) means that the operation consumes no operand or evaluates to nothing.

For each type there is a corresponding predicate operation (with the same name but ending with parenthesis) that expects an object and determines whether the given object is of such type. For example, for the type **number** there is the predicate **number?**. The type **any** means any type.

+ *(number∗ → number)*
: Addition.

∗ *(number∗ → number)*
: Multiplication.

- *(number+ → number)*
: Subtraction.

/ *(number+ → number)*
: Division.

= *(number × number → boolean)*
: Number equivalency.

< *(number × number → boolean)*
: Lesser than.

> *(number × number → boolean)*
: Greater than.

boolean? *any → boolean)*
: Predicate for the boolean type.

byte? *any → boolean)*
: Predicate for the byte type.

current-input-port *(∅ → port)*
: The standard input port.

current-error-port *(∅ → port)*
: The standard error port.

current-output-port *(∅ → port)*
: The standard output port.

define *(symbol × any → ∅)*
: Binds symbol to object in current environment.

display *(any × output-port? → ∅)*
: Print user-readable form of object into port (default to current output port).

false *(∅ → false)*
: False.

false? *(any → boolean)*
: Predicate for false.

lambda *(vector × any → closure)*
: Creates applicative procedure closure. The given vector must contain zero or more symbols, which are used as formal parameters for the procedure.

macro *(vector × any → closure)*
: Creates non-applicative procedure closure. The given vector must contain one or more symbols,

which are used as formal parameters for the procedure.

make-environment *(environment? → environment)*
Creates a new environment. If an environment is provided, uses it as parent environment, other wise the created environment is orphan.

make-string *(n:size × c:byte? → string)*
Return newly allocated string of size `n` filled with either `c` or the zero byte.

make-vector *(n:size × o:any? → string)*
Return newly allocated vector of size `n` filled with either `o` or the null element.

newline *(output-port? → ∅)*
Print newline into port (default to current output port).

null? *(any → boolean)*
Predicate for the empty vector.

not *(any → boolean)*
Negation.

port? *(any → boolean)*
Predicate for port.

quote *(any → any)*
Quote object.

same? *(any any → boolean)*
Whether two objects are the same.

set! *(symbol × any → ∅)*
If the symbol is bound in the current environment, rebound it to the given object; signals an error otherwise

string-compare *(s0:string × s1:string → number)*
Returns an integer less than, equal to, or greater than 0, according as `s0` is lexicographically less than, equal to, or greater than `s1`.

string? *(any → boolean)*
Predicate for string.

string-equiv? *(s0:string × s1:string → boolean)*
Return whether `s0` and `s1` are equivalent.

stirng-ref *(s:string × i:number → byte)*
Return the `i`-th element of `s`. It is an error if `s` has no `i` element.

string-set! *(s:string × i:number × b:byte → ∅)*
Set the `i`-th element of `s` to `b`. It is an error if `s` has no `i` element.

string-copy *(string → string)*
Return newly allocated string with elements from given string.

string-length *(string → number)*
Return number of bytes in the given string.

string->vector *(string → vector)*
Return newly allocated vector of bytes from given string.

symbol? *(any → boolean)*
Predicate for symbol.

true *(∅ → false)*
True.

true? *(any → boolean)*
Predicate for true.

void *(∅ → ∅)*
Nothing.

vector *(any* → vector)*

Create vector with given elements.

vector-copy *(string → string)*
Return newly allocated vector with elements from given vector.

vector-length *(vector → number)*
Returns the number of bytes in the given vector.

vector-ref *(v:vector × i:number → any)*
Returns the *i*-nth element of *v*. It is an error if *v* has no *i* element.

vector-set! *(v:vector × i:number × o:any → ∅)*
Set the *i*-nth element of *v* to *o*. It is an error if *v* has no *i* element.

wrap *(operative → applicative)*
Converts operative to applicative.

write *(any × output-port? → ∅)*
Print printed external representation of object into port (default to current output port).

FORMAL SYNTAX

This section provides a formal syntax for **simp** written in an extended Backus-Naur form (BNF). Nonterminals are written between angle braces (**<...>**). A terminal symbol is written between double quotation marks (**"..."**).

The following extensions to BNF are used to make the description more consise:

- **<thing>*** means zero or more occurrences of **<thing>**.
- **<thing>+** means one or more occurrences of **<thing>**.
- **<thing>?** means zero or one occurrence of **<thing>**.

The BNF is augmented with the concepts of character classes and character ranges. A *character class* is expressed between square braces and colons (**[:...:]**) and denotes a named set of characters. A *character range* is a set of characters and/or character classes between square braces (**[...]**) and denotes any character in the set or in the classes. For example, (**[abc[:delimiter:]]**) means an **a**, or **b**, or **c** character, or a character in the **[:delimiter:]** class. The notion of character range is augmented as follows.

- The **-** character has the same special meaning in a character range it has in ERE. For example, **[0-9]** is the same as **[0123456789]** (which is the same as **[:decimal:]**).
- The **^** character has the same special meaning in a character range it has in ERE. For example, **[^abc]** means any character but **a**, **b**, or **c**.
- The opening bracket **[** may occur anywhere in a character range.
- The closing bracket **]** may occur only as the first character in a character range

Unprintable and hard-to-type characters are represented in the same escape notation used in string literals. For example, **\n** is the newline.

Alphabet

The alphabet for this grammar is all the 256 bytes that can be read from a file augmented with the end-of-file indicator.

The character classes are defined as follows.

```
[:space:]          ← [ \f\n\r\t\v]
[:binary:]         ← [0-1]
[:octal:]          ← [0-7]
[:decimal:]        ← [0-9]
[:hexadecimal:]    ← [0-9A-Fa-f]
```

```
[:delimiter:]      ← [][()#[:eof:][:space:]]
```

The end-of-file indicator, in special, is represented by the special class **[:eof:]**.

The backslash character (****), the double-quote character (**"**), and the single-quote character (**'**), which have special meanings and thus would need to be escaped, are represented by the special classes **[:slash:]**, **[:double-quote:]**, and **[:single-quote:]**, respectively.

The special character class **[:anything:]** represents any character in the alphabet.

Tokens

A **token** is the lexical element used to compose well formed expressions. Some characters, known as **delimiters**, have special meaning during the program parsing, because certain tokens require a delimiter to occur after them. A token is defined as follows:

```
<token>            ← <end-of-file>
                   | <left-paren>
                   | <right-paren>
                   | <identifier>
                   | <char-literal>
                   | <string-literal>
                   | <number-literal>
```

The end-of-file is the token that terminates a program. It is actually not a character, but is interpreted as if it were.

```
<end-of-file>      ← [:eof:]
```

Single-character tokens are the following:

```
<left-paren>       ← "("
<right-paren>      ← ")"
```

A character literal is composed by one character element between single quotes. A string literal is composed by zero or more character elements between double quotes. A character element is any character other than a double quote or a backslash or an escaped character. Chracter literals and string literals are used to represent characters and strings (alsow known as bytevectors) respectively. The single and the double quotation mark characters that terminates a character and a string are themselves delimiters.

```
<char-literal>    ← [:single-quote:] <string-element> [:single-quote:]
<string-literal>  ← [:double-quote:] <string-element>* [:double-quote:]
<string-element>  ← [^[:double-quote:][:slash:]]
                  | [:slash:][:anything:]
```

A number literal begins with an optional signal and is followed by the number body. A delimiter must occur after a numer literal.

```
<number-literal>  ← <signal> <number-body>
<signal>          ← [+-]?
<number-body>     ← <binary-literal>
                  | <octal-literal>
                  | <decimal-literal>
                  | <hex-literal>
                  | <real-literal>
<binary-literal>  ← 0 [bB] [[:binary:]]*
<octal-literal>   ← 0 [oO] [[:octal:]]*
<decimal-literal> ← 0 [dD] [[:decimal:]]*
<hex-literal>     ← 0 [dD] [[:hexadecimal:]]*
```

```
<real-literal>     ← [[:decimal:]]+ <fraction>? <exponent>?
<fraction>         ← "." [[:decimal:]]*
<exponent>         ← <signal> [[:decimal:]]*
```

An identifier is any sequence of non-delimiter characters that does not form another type of token. A delimiter character must occur after an identifier.

```
<identifier>       ← <initial> [^[:delimiter:]]*
<initial>          ← "+" [^[:decimal:][:delimiter:]]
                   | "-" [^[:decimal:][:delimiter:]]
                   | [^+-.[:decimal:][:delimiter:]]
```

Escape sequences

Within a string literal, sequences of characters beginning with a backslash (****) are called **escape sequences** and represent bytes other than the characters themselves. Most escape sequences represent a single byte, but some forms may represent more than one byte. An invalid escape sequence is equivalent to the character after the backslash; for example, the string literal “**\j**” does not contain a valid escape sequence, so it is equivalent to “**j**”. The valid escape sequences are as follows:

\a Alarm (U+0007).
\b Backspace (U+0008).
\t Horizontal tab (U+0009).
\n Line feed (U+000A).
\v Vertical tab (U+000B).
\f Form feed (U+000C).
\r Carriage return (U+000D).
\e Escape character (U+001B).
\" Double quote (U+0022).
\\ Backslash (U+005C).
num Byte whose value is the 1-, 2-, or 3-digit octal number *num*.
\x*num* Byte whose value is the 1- or 2-digit hexadecimal number *num*.
\u*num* Bytes encoding, in UTF-8, the 4-digit hexadecimal number *num*.
\U*num* Bytes encoding, in UTF-8, the 8-digit hexadecimal number *num*.

Intertoken space

Tokens are separated by intertoken space, which includes both whitespace and comments. Intertoken space is used for improved readability, and as necessary to separate tokens from each other.

```
<whitespace>       ← [[:space:]]
<comment>          ← "#" [^\n]* "\n"
<atmosphere>       ← <whitespace> | <comment>
<intertoken>       ← <atmosphere>*
```

Whitespace can occur between any two tokens, but not within a token. Whitespace occurring inside a string literal is significant.

Comments are annotations in the source code and are treated exactly like whitespace. A hash character (**#**) outside a string literal indicates the start of a comment. The comment continues to the end of the line on which the hash character appears.

Read external representation

The following is a simplification of the syntax of a read external representation. This syntax is not complete, because intertoken-space may occur on either side of any token (but not within a token).

```
<representation>   ← <number>
                   | <string>
```

```
                    | <symbol>
                    | <vector>
<number>            ← <number-literal>
<byte>              ← <char-literal>
<string>            ← <string-literal>
<symbol>            ← <identifier>
<vector>            ← <left-paren> <representation>* <right-paren>
```

Program

A **simp** program is a sequence of characters forming whitespace, comments, and tokens. The tokens in a program must form syntactically well formed expressions.

```
<program>           ← <expression>*
<expression>        ← <variable>
                    | <literal>
                    | <application>
<variable>          ← <symbol>
<literal>           ← <number> | <string>
<application>       ← <vector>
```

FORMAL SEMANTICS

I have no idea what a formal semantics is or does.

EXAMPLES

[TODO]

SEE ALSO

simp(1), simp(3)

Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programms*, *The MIT Press*, 1996.

STANDARDS

The **simp** programming language is compliant with nothing, as it has not been standardised. It was influenced by the Scheme and Kernel LISP dialects.

The syntax for comments and number literals breaks the usual LISP tradition, and are influenced by shell script comments and C constants, respectively.

Parts of this manual (especially at the **DESCRIPTION** section) were blatantly stolen from *Revised Report on the Algorithmic Language Scheme*.

HISTORY

The **simp** programming language was developed as a personal playground for programming language theory, motivated by the reading of the Wizard Book (Abelson & Sussman). It first appeared as a C library in 2022.

AUTHORS

The **simp** programming language was designed by Lucas de Sena <lucas AT seninha DOT org>.

BUGS

The **simp** programming language implemented in `simp`(1) and `simp`(3) is not complete, and may not conform to this manpage.

This manual page is also not complete, as the language is only informally specified, and may change significantly from one release to the other.

This manual uses the terms "string" and "bytevector" interchangeably, as both refer to the same **simp** data structure. Note that "string" and "string literal" refer to different concepts; the former is a data type, while the latter is a token type.

This manual avoids to use the word "character" to refer to the elements of a string. This manual uses the word "character" to refer solely to the units that compose tokens read by the parser. Strings in **simp** can possibly contain no valid character (in the sense of a UTF-8 encoded codepoint). This manual uses the term "byte" instead to refer to the elements of a string.

There's no "character" data type, either in the C sense of a "byte", or in the sense of a UTF-8 encoded codepoint. A single byte can be represented as a one-element string. A UTF-8 encoded codepoint can be represented as a string containing the encoding bytes.