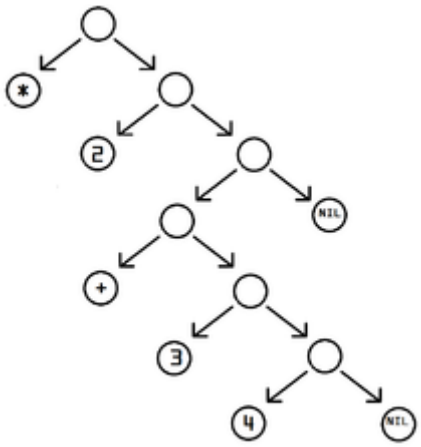
S-expression

In computer programming, **S-expressions** (or **symbolic expressions**, abbreviated as **sexprs**) are a notation for nested list (tree-structured) data, invented for and popularized by the programming language Lisp, which uses them for source code as well as data. In the usual parenthesized syntax of Lisp, an S-expression is classically defined[1] as

1. an atom, or
2. an expression of the form (*x* . *y*) where *x* and *y* are S-expressions.

The second, recursive part of the definition represents an ordered pair, which means that S-expressions can represent any binary tree, though S-expressions which contain cycles cannot conversely be represented as binary trees.



Tree data structure representing the S-expression `(* 2 (+ 3 4))`

The definition of an atom varies per context; in the original definition by John McCarthy,[1] it was assumed that there existed "an infinite set of distinguishable atomic symbols" represented as "strings of capital Latin letters and digits with single embedded blanks" (i.e., character string and numeric literals). Most modern sexpr notations in addition use an abbreviated notation to represent lists in S-expressions, so that

```
(x y z)
```

stands for

```
(x . (y . (z . NIL)))
```

where `NIL` is the special end-of-list object (alternatively written `()`, which is the only representation in Scheme[2]).

In the Lisp family of programming languages, S-expressions are used to represent both source code and data. Other uses of S-expressions are in Lisp-derived languages such as DSSSL, and as mark-up in communication protocols like IMAP and John McCarthy's CBCL. It's also used as text representation of WebAssembly. The details of the syntax and supported data types vary in the different languages, but the most common feature among these languages is the use of S-expressions and prefix notation.

Contents

Datatypes and syntax

There are many variants of the S-expression format, supporting a variety of different syntaxes for different datatypes. The most widely supported are:

- *Lists and pairs*: `(1 () (2 . 3) (4))`
- *Symbols*: `with-hyphen ?@!$ a\ symbol\ with\ spaces`
- *Strings*: `"Hello, world!"`
- *Integers*: `-9876543210`
- *Floating-point numbers*: `-0.0 6.28318 6.022e23`

The character # is often used to prefix extensions to the syntax, e.g. `#x10` for hexadecimal integers, or `#\C` for characters.

Use in Lisp

When representing source code in Lisp, the first element of an S-expression is commonly an operator or function name and any remaining elements are treated as arguments. This is called "prefix notation" or "Polish notation". As an example, the Boolean expression written `4 == (2 + 2)` in C, is represented as `(= 4 (+ 2 2))` in Lisp's s-expr-based prefix notation.

As noted above, the precise definition of "atom" varies across LISP-like languages. A quoted string can typically contain anything but a quote, while an unquoted identifier atom can typically contain anything but quotes, whitespace characters, parentheses, brackets, braces, backslashes, and semicolons. In either case, a prohibited character can typically be included by escaping it with a preceding backslash. Unicode support varies.

The recursive case of the s-expr definition is traditionally implemented using cons cells.

S-expressions were originally intended only for data to be manipulated by M-expressions, but the first implementation of Lisp was an interpreter of S-expression encodings of M-expressions, and Lisp programmers soon became accustomed to using S-expressions for both code and data. This means that Lisp is homoiconic; that is, the primary representation of programs is also a data structure in a primitive type of the language itself.

Examples of data S-expressions

Nested lists can be written as S-expressions: `((milk juice) (honey marmalade))` is a two-element S-expression whose elements are also two-element S-expressions. The whitespace-separated notation used in Lisp (and this article) is typical. Line breaks (newline characters) usually qualify as separators.

This is a simple context-free grammar for a tiny subset of English written as an S-expression (Gazdar/Melish, Natural Language Processing in Lisp), where S=sentence, NP=Noun Phrase, VP=Verb Phrase, V=Verb:

```
(((S) (NP VP))
 ((VP) (V))
 ((VP) (V NP))
 ((V) died)
 ((V) employed)
 ((NP) nurses)
 ((NP) patients)
 ((NP) Medicenter)
 ((NP) "Dr Chan"))
```

Example of source code S-expressions

Program code can be written in S-expressions, usually using prefix notation.

Example in Common Lisp:

```
(defun factorial (x)
   (if (zerop x)
       1
       (* x (factorial (- x 1)))))
```

S-expressions can be read in Lisp using the function READ. READ reads the textual representation of an S-expression and returns Lisp data. The function PRINT can be used to output an S-expression. The output then can be read with the function READ, when all printed data objects have a readable representation. Lisp has readable representations for numbers, strings, symbols, lists and many other data types. Program code can be formatted as pretty printed S-expressions using the function PPRINT (note: with two Ps, short for *pretty*-print).

Lisp programs are valid S-expressions, but not all S-expressions are valid Lisp programs. `(1.0 + 3.1)` is a valid S-expression, but not a valid Lisp program, since Lisp uses prefix notation and a floating point number (here 1.0) is not valid as an operation (the first element of the expression).

An S-expression preceded by a single quotation mark, as in `'x`, is syntactic sugar for a quoted S-expression, in this case `(quote x)`.

Parsing

S-expressions are often compared to XML: the key difference is that S-expressions have just one form of containment, the dotted pair, and are much easier to parse, while XML tags can contain simple attributes, other tags, or CDATA, each using different syntax.

Standardization

Standards for some Lisp-derived programming languages include a specification for their S-expression syntax. These include Common Lisp (ANSI standard document ANSI INCITS 226-1994 (R2004)), Scheme (R5RS and R6RS[3]), and ISLISP.

Rivest's variant

In May 1997, Ron Rivest submitted an Internet-Draft[4] to be considered for publication as an RFC. The draft defined a syntax based on Lisp S-expressions but intended for general-purpose data storage and exchange (similar to XML) rather than specifically for programming. It was never approved as an RFC, but it has since

been cited and used by other RFCs (e.g. RFC 2693) and several other publications.[5] It was originally intended for use in SPKI.

Rivest's format defines an S-expression as being either an octet-string (a series of bytes) or a finite list of other S-expressions. It describes three interchange formats for expressing this structure. One is the "advanced transport", which is very flexible in terms of formatting, and is syntactically similar to Lisp-style expressions, but they are not identical. The advanced transport, for example, allows octet-strings to be represented verbatim (the string's length followed by a colon and the entire raw string), a quoted form allowing escape characters, hexadecimal, Base64, or placed directly as a "token" if it meets certain conditions. (Rivest's tokens differ from Lisp tokens in that the former are just for convenience and aesthetics, and treated exactly like other strings, while the latter have specific syntactical meaning.)

Rivest's draft defines a canonical representation "for digital signature purposes". It's intended to be compact, easier to parse, and unique for any abstract S-expression. It only allows verbatim strings, and prohibits whitespace as formatting outside strings. Finally there is the "basic transport representation", which is either the canonical form or the same encoded as Base64 and surrounded by braces, the latter intended to safely transport a canonically encoded S-expression in a system which might change spacing (e.g. an email system which has 80-character-wide lines and wraps anything longer than that).

This format has not been widely adapted for use outside of SPKI (some of the users being GnuPG, libgcrypt, Nettle, and GNU lsh). Rivest's S-expressions web page provides C source code for a parser and generator (available under the MIT license), which could be adapted and embedded into other programs.[6] In addition, there are no restrictions on independently implementing the format.

See also

- cons
- CAR and CDR
- Fexpr
- Lambda calculus
- M-expression
- Canonical S-expressions
- Comparison of data serialization formats

References

1. John McCarthy (1960/2006). Recursive functions of symbolic expressions (http://www-formal.stanford.edu/jmc/recursive/recursive.html) Archived (https://web.archive.org/web/20040202215021/http://www-formal.stanford.edu/jmc/recursive/recursive.html) 2004-02-02 at the Wayback Machine. Originally published in Communications of the ACM.
2. "Revised^5 Report on the Algorithmic Language Scheme" (https://schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-9.html). *schemers.org*.
3. Sperber, Michael; Dybvig, R. Kent; Flatt, Matthew; Van Straaten, Anton; Findler, Robby; Matthews, Jacob (Aug 12, 2009). "Revised6 Report on the Algorithmic Language Scheme". *Journal of Functional Programming*. **19** (S1): 1–301. CiteSeerX 10.1.1.372.373 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.372.373). doi:10.1017/S0956796809990074 (https://doi.org/10.1017%2FS0956796809990074).
4. S-expressions (http://people.csail.mit.edu/rivest/Sexp.txt), Network Working Group, Internet Draft, Expires November 4, 1997 - R. Rivest, May 4, 1997 draft-rivest-sexp-00.txt, Ronald L. Rivest, CSAIL MIT website

5. rivest sexp (https://scholar.google.com/scholar?hl=en&lr=&safe=off&q=rivest+sexp&btnG=Search), Google Scholar (search)
6. "SEXP (S-expressions)" (http://people.csail.mit.edu/rivest/sexp.html). *people.csail.mit.edu*.

External links

- sfsexp (https://github.com/mjsottile/sfsexp) the small, fast S-expression library for C/C++ on Github
- minilisp (http://leon.bottou.org/projects/minilisp), by Léon Bottou.
- S-expressions on Rosettacode (http://rosettacode.org/wiki/S-expressions) has implementations of readers and writers in many languages.
