

LISP PROGRAM-SIZE COMPLEXITY IV

Applied Mathematics and Computation
52 (1992), pp. 141–147

G. J. Chaitin

Abstract

We present a new “character-string” oriented dialect of LISP in which the natural LISP halting probability asymptotically has maximum possible LISP complexity.

1. Introduction

This paper continues the study of LISP program-size complexity in my monograph [1, Chapter 5] and the series of papers [2–4].

In this paper we consider a dialect of LISP half-way between the

Copyright © 1992, Elsevier Science Publishing Co., Inc., reprinted by permission.

LISP considered in my paper [2] (which is essentially normal LISP [5]) and that studied in my monograph [1]. The “character-string” LISP studied in this paper employs multiple-character atoms as in [2], but allows all possible character strings as S-expressions subject only to the requirement that parentheses balance as in [1]. Precise counts of S-expressions as in [1] rather than subadditivity arguments as in [2], are used to show that the maximum possible character-string LISP complexity $H_{cs}(s)$ of an n -bit string s is $\beta n + O(\log n)$, where $\beta = 1/\log_2 \alpha$ and $\alpha =$ the number of characters in the alphabet. A particularly natural definition of a character-string LISP halting probability Ω_{cs} is presented here. The n -bit initial segment of the base-two expansion of Ω_{cs} asymptotically achieves maximum possible character-string LISP complexity βn . Indeed, the entire theory developed in [1, Section 5.1] for toy LISP and in [2, 3] for H_{LISP} can be reformulated in terms of H_{cs} .

2. Précis of Character-String LISP

Let’s put the LISP of [5] on the operating table and examine it from an information-theoretic point of view. The problem is that sometimes different looking S-expressions produce the same internal structure when read in by the LISP interpreter and look the same if then written out. In other words, the problem is **synonyms!** Information is being wasted because not all different strings of characters in the external representation of an S-expression lead to different S-expressions in the internal representation, which consists of binary trees of pointers.

In my monograph [1] I fixed this by using drastic surgery. First of all, blanks are eliminated and all atoms are exactly one character long. Next, `()` and `nil` are no longer synonyms, because the only way to denote the empty list is via `()`. And “true” and “false” are `1` and `0` instead of `t` and `nil`.

The illness is serious, but the cure in [1] is rather drastic. Here we present another way to eliminate synonyms and improve the expressive power of LISP from an information-theoretic point of view. This time the aim is to keep things as much as possible the way they are in normal LISP [5]. So each extra, not-strictly-necessary blank is now a “blank

atom,” which prints as a blank or `␣` when we want to show it. And both `nil` and `()` denote an empty list, but will print out differently and compare different internally using the LISP primitive function `eq`.

Also we allow **any** succession of characters in a legal S-expression between an opening left parenthesis and a closing right parenthesis, and consider that each such S-expression **is different**. The only rule is that parentheses must balance.

I think that a good way to express this idea is to call it a “character-string” oriented version of LISP. It might also be called a “wysiwyg” (“what you see is what you get”) version of LISP. The external representation of S-expressions is now taken seriously; before only the internal representation of S-expressions really counted.

In summary, internal and external format are now precisely equivalent, and reading an S-expression in and then writing it out gives exactly the same thing back that was read in. There are no synonyms: `()` and `nil` are different, `00022` and `22` are different, and `(x_ly)` and `(x_l_l_l_y)` are different. Each different string of characters is a different S-expression and blanks within an S-expression are respected. `0x0x` is a valid atom.

For example, the following S-expressions **are all different**:

```
(a_l_b_nil)
(a_l_b_l())
(a_l_b())
(a_l_b_l_l())
(a_l_b_l())
(a_l_b_l_l_l())
(a_l_b_l_l())
(l_a_l_b_l())
(l_a_l_b())
(l_a_l_b_l_nil)
```

(a␣␣␣b␣␣␣nil)

(a␣␣␣b␣␣␣())

In normal LISP, these would all be the same S-expression. And the LISP primitive-function `eq` is defined to be character-string equality, so that it can see that all these S-expressions are different.

But what do these S-expressions with blanks mean? Consider the following character-string LISP S-expression:

(␣␣␣abc␣␣␣def␣␣␣)

The first three blanks denote three blank atoms. The second three blanks denote only two blank atoms. And the last three blanks denote three blank atoms. Blank atoms are always **a single blank**.

In general, n consecutive blanks within an S-expression will either denote n blank atoms or $n - 1$ blank atoms; it will be $n - 1$ blank atoms iff the blanks separate two consecutive atoms $\neq()$. In other words, n consecutive blanks denote n blank atoms except when they separate two characters that are neither parentheses nor blanks:

...abc␣␣␣def...

In this situation, and only this situation, $n + 1$ blanks denote n blank atoms. E.g., a single blank separating two atoms that are $\neq()$ does not entail any blank atoms:

...abc␣def...

On the other hand, here there is no blank atom:

... (abc) (def) ...

And here there is exactly one blank atom:

... (abc)␣(def) ...

With this approach, `append` of two S-expressions will carry along all the blanks in its operands, and will add a blank if the first list ends with an atom $\neq()$ and the second list begins with an atom $\neq()$:

$$\$(\text{append}('(_a_b_c_))('(_d_e_f_))) \rightarrow (_a_b_c_d_e_f_)$$

$$\$(\text{append}('(_a_b_c_))('(_d)_e_f_))) \rightarrow (_a_b_c_(_d)_e_f_)$$

Thus the result of appending a list of n elements and a list of m elements always has $n+m$ elements as usual. $\$$ starts a mode in which all blanks are significant in the next complete S-expression. In the normal mode, excess blanks are removed, as is usual in LISP.

A few more words on $\$$: Normally, extra blanks are helpful in writing S-expressions, to indicate their structure, but we don't really want the blank atoms. So we use the LISP input "meta-character" $\$$ to indicate that in the next complete S-expression extra blanks should not be removed. For example, $\$_$ is a single blank atom. And $\$(____)$ is a list of three blank atoms. But the input expression

$$(a____\$(________)b)$$

denotes

$$(a(________)b)$$

because the extra blanks next to the a and the b are outside the range of the $\$$. The use of $\$$ as a meta-character makes it impossible to have $\$$'s in the name of an atom.

Note that when taking `car`, `cdr` and `cons`, blanks are usually just carried along, but sometimes a single blank must be stripped off or added. The following examples explain how this all works:

- $\$(\text{car}('(_a_b_c_))) \rightarrow _$
 $\$(\text{cdr}('(_a_b_c_))) \rightarrow (a_b_c_)$
 $\$(\text{cons}(')_)('(a_b_c_))) \rightarrow (_a_b_c_)$
- $\$(\text{cons}('a)('(b_c_))) \rightarrow (a_b_c_)$
 (adds one blank after the a)
 $\$(\text{cons}('a)('(_b_c_))) \rightarrow (a_b_c_)$
 (adds one blank after the a)
 $\$(\text{cdr}('(_a_b_c_))) \rightarrow (_b_c_)$
 (eliminates one blank after the a)

- $\$(\text{cons}('a)('((b)_ _ c))) \rightarrow (a(b)_ _ c)$
 (doesn't add one blank after the a)
- $\$(\text{cons}('a)('(_ (b)_ _ c))) \rightarrow (a_ (b)_ _ c)$
 (doesn't add one blank after the a)
- $\$(\text{cdr}('a_ (b)_ _ c))) \rightarrow (_ (b)_ _ c)$
 (doesn't eliminate one blank after the a)
- $\$(\text{cons}('a)('nil)) \rightarrow (a)$
 $\$(\text{cons}('a)('())) \rightarrow (a)$
 $\$(\text{cdr}('a))) \rightarrow \text{nil}$
 (cdr never yields ())
- $\$(\text{eq}('nil)('())) \rightarrow \text{nil}$

The primitive functions `car`, `cdr` and `cons` are defined so that if there are no extra blanks, they give exactly the same value as in normal LISP. The primitive-function `eq` is defined to be character-string equality of entire S-expressions, not just atoms. Of course, the convert S-expression to character-string primitive function (see [2, Section 2]) gives each repeated blank; this and `eq` are a way to extract all the information from an S-expression. And the convert character-string to S-expression primitive function (see [2, Section 2]) is able to produce all possible S-expressions.

In summary, this new approach avoids the fact that `()` is the same as `nil` and blanks are often ignored in LISP. We now allow and distinguish all combinations of blanks; this makes it possible to pack much more information in an S-expression. We must also allow names of atoms with any possible mix of characters, as long as they are neither blanks nor parentheses; we cannot outlaw, as normal LISP does, the atom `9xyz`. In normal LISP [5], if an atom begins with a digit, it must **all** be digits. In our LISP, `9xyz` is allowed as the name of an atom. Thus our definition of an integer is that it is an atom that is all digits, possibly preceded by a minus sign (hyphen), not an atom that begins with a digit. Also we must allow numbers with 0's at the left like `00022`, and `eq` must consider `00022` and `022` to be different. There is a different equality predicate “=” that is for numbers.

Note that it is still possible to implement character-string LISP efficiently via binary trees of pointers as is done in normal LISP. Efficient implementations of normal LISP are based on cells containing two pointers, to the `car` and the `cdr` of an S-expression (see [5]).

3. The Halting Probability Ω_{cs}

Of course the character-string LISP complexity $H_{cs}(x)$ of an S-expression x is now defined to be the minimum size in characters $|e|$ of a character-string LISP S-expression e that evaluates to x . Here e is in the “official” character-string LISP notation in which every blank is significant, not in the “meta-notation” used in Section 2 in which only blanks in the range of a \$ are significant.

The new idea that we presented in Section 2 is to think of LISP S-expressions as character strings, to allow **any** succession of characters in a legal S-expression between an opening left parenthesis and a closing right parenthesis, and to consider that each such S-expression **is different**. The only rule is that parentheses must balance. From the discussion of toy LISP in [1], we know that having parentheses balance does not significantly decrease the multiplicative growth of the number of possibilities. I.e., the number of S-expressions with n characters has a base-two logarithm that is asymptotic to $n \log_2 \alpha$, where α is the number of characters in the LISP alphabet including the blank and both parentheses. (See [1, Appendix B].)

More precisely, the analysis of [1, Appendix B] gives the exact number and the asymptotics of the **non-atomic** character-string LISP S-expressions of size n . There are also $(\alpha - 3)^n$ **atomic** character-string LISP S-expressions of size n , which is negligible in comparison. Thus the total number S_n of character-string LISP S-expressions with exactly n characters is asymptotic to

$$S_n \sim \frac{\alpha^{n-2}}{2\sqrt{\pi}(n/\alpha)^{1.5}}.$$

Hence

$$\log_2 S_n = n \log_2 \alpha + O(\log n).$$

From this it is easy to see that the maximum possible character-string LISP complexity $H_{cs}(s)$ of an n -bit string s is $\beta n + O(\log n)$, where $\beta = 1/\log_2 \alpha$ and $\alpha =$ the number of characters in the alphabet (including the blank and both parentheses).

We see that the rules in Section 2 remove almost all the redundancy in normal LISP S-expressions.¹ Also, because no extension of a non-atomic S-expression (e) is a valid non-atomic S-expression, we can define a character-string LISP halting probability Ω_{cs} as follows:

$$\Omega_{cs} = \sum_{(e) \text{ has a value}} \alpha^{-[\text{size of } (e)]} = \sum_{(e) \text{ halts}} \alpha^{-|(e)|}.$$

Just as in [3, Section 4], we see that being given the first $n + O(\log n)$ bits of the base-two expansion of Ω_{cs} would enable one to determine the character-string LISP complexity H_{cs} of each $\leq n$ bit string. The maximum of $H_{cs}(s)$ taken over all $\leq n$ bit strings s is asymptotic to βn . As in [3, Section 4], it follows that the string consisting of the first n bits of the base-two expansion of Ω_{cs} itself asymptotically has maximum possible character-string LISP complexity $H_{cs} \sim \beta n$.² Thus we can follow [3, Section 5] and construct from Ω_{cs} diophantine equations D_1 and D_2 with the following property: To answer either the first n cases of the yes/no question Q_1 in [3, Section 5] about equation D_1 or the first n cases of the yes/no question Q_2 in [3, Section 5] about equation D_2 requires a formal system with character-string LISP complexity $> \beta n + o(n)$. I.e., the proof-checking function associated with a formal system that enables us to determine the first n bits of the base-two expansion of Ω_{cs} must have character-string LISP complexity $> \beta n + o(n)$.

References

- [1] G. J. CHAITIN, *Algorithmic Information Theory*, 3rd Printing, Cambridge: Cambridge University Press (1990).

¹Hence minimal character-string LISP S-expressions for a given result are essentially unique, and can also be proved to be “normal” (i.e., in the limit, all α possible characters occur with equal relative frequency). See [1, Section 5.1].

²From this it is not difficult to show that Ω_{cs} is BOREL normal in every base. See [3, Section 9].

- [2] G. J. CHAITIN, “LISP program-size complexity,” *Applied Mathematics and Computation* **49** (1992), 79–93.
- [3] G. J. CHAITIN, “LISP program-size complexity II,” *Applied Mathematics and Computation*, in press.
- [4] G. J. CHAITIN, “LISP program-size complexity III,” *Applied Mathematics and Computation*, in press.
- [5] J. MCCARTHY et al., *LISP 1.5 Programmer’s Manual*, Cambridge MA: MIT Press (1962).