

LISP PROGRAM-SIZE COMPLEXITY III

Applied Mathematics and Computation
52 (1992), pp. 127–139

G. J. Chaitin

Abstract

We present a “parenthesis-free” dialect of LISP, in which (a) each primitive function has a fixed number of arguments, and (b) the parentheses associating a primitive function with its arguments are implicit and are omitted. The parenthesis-free complexity of an S-expression e is defined to be the minimum size in characters $|p|$ of a parenthesis-free LISP expression p that has the value e . We develop a theory of program-size complexity for parenthesis-free LISP by showing (a) that the maximum possible parenthesis-free complexity of an n -bit string is $\sim \beta n$, and (b) how to construct three parenthesis-free LISP halting probabilities Ω_{pf} , Ω'_{pf} and Ω''_{pf} .

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

1. Introduction

In this paper we consider a dialect of LISP half-way between the LISP considered in my paper [1] (which is essentially normal LISP [2]) and that studied in my monograph [3]. The “parenthesis-free” LISP studied in this paper employs multiple-character atoms as in [1], but omits the parentheses associating each primitive function with its arguments as in [3]. Subadditivity arguments as in [1] rather than precise counts of S-expressions as in [3], are used to show that the maximum possible parenthesis-free LISP complexity H_{pf} of an n -bit string is $\sim \beta n$. A particularly natural definition of a parenthesis-free LISP halting probability Ω_{pf} is presented here. Also two other halting probabilities, Ω'_{pf} and Ω''_{pf} , that asymptotically achieve maximum possible parenthesis-free LISP complexity βn of the n -bit initial segments of their base-two expansions. We thus show that the entire theory developed in [1, 4] for H_{LISP} can be reformulated in terms of H_{pf} .

In the last section we make our parenthesis-free LISP substantially easier to use.¹

2. Précis of Parenthesis-Free LISP

Let’s start with an example! Here is a sample specimen of parenthesis-free LISP. It is a self-contained LISP expression that defines a function **append** for concatenating two lists and then applies this function to the lists (a b c) and (d e f).

Old notation [2]: see Figure 1. This expression evaluates to (a b c d e f). New notation: see Figure 2. This expression, which is what we shall call a meta-expression or M-expression, is expanded as it is read by the parenthesis-free LISP interpreter into the corresponding S-expression, which has all the parentheses: see Figure 3. This expression also evaluates to (a b c d e f).

In parenthesis-free LISP, each LISP primitive function must have a fixed number of arguments. So we have to fix **cond**, **define**, **plus**,

¹Two decades ago the author wrote an interpreter for a similar LISP dialect. At that time he did not realize that a mathematical theory of program size could be developed for it.

```
(
  (lambda (append)
    (append (quote (a b c)) (quote (d e f)))
  )
  (quote
    (lambda (x y)
      (cond ((atom x) y)
            (t (cons (car x) (append (cdr x) y))))
    )
  )
)
)
```

Figure 1. Old Notation

```
(fnc (app) (app '(a b c) '(d e f))
'fnc (x y)
  if at x y
    jn hd x (app tl x y)
)
```

Figure 2. M-expression

```
((fnc (app) (app ('(a b c)) ('(d e f))))
('fnc (x y)
  (if (at x) y
      (jn (hd x) (app (tl x) y))))
)
```

Figure 3. S-expression

and `times`. We replace conditional expressions with an indefinite number of arguments by if-then-else expressions with three arguments. And `define`, `plus`, and `times` now always have two arguments. As long as we're at it, we also shorten the names of the primitive functions. See the table summarizing parenthesis-free LISP on page 5.

Functions dealing with integers are the same as in C. `mexp` converts an S-expression into the list of characters in the smallest possible equivalent M-expression.² This will be a list of integers in the range from 0 to $\alpha - 1$ (α = the size of the alphabet). `sexp` converts the list of characters in an M-expression into the corresponding S-expression.³ `$` indicates that there are no implicit parentheses in the immediately following S-expression. (`$` also loses any special meaning within the range of a `$`.) Thus

'`$(hd t1 jn $)`

evaluates to `(hd t1 jn $)`. Another detail: `'` and `$` do not need a blank before the next character, i.e., no other atoms can start with the characters `'` or `$`.

For this parenthesis-free approach to work, it is important that

- (a) Every S-expression can be written in this notation.
- (b) It should be possible given a parenthesis-free LISP S-expression to calculate the equivalent M-expression⁴ **of smallest size**. Here is an example of the synonym problem: `'$(eq x at y)` and `'($eq x $at y)` are the same.
- (c) And one must be able to calculate the size in characters of the M-expression that corresponds to a given S-expression as in (b).

3. Parenthesis-Free LISP Complexity

Previously we had two LISP theories of program-size complexity: one for real LISP [1, 4], and one for a toy LISP [3]. In this section we

²This is the inverse of what the LISP interpreter's read routine does.

³This is an internally available version of the read routine used by the LISP interpreter.

⁴This is the inverse of the input parse that puts in the implicit parentheses.

Old Name	New Name	Read As	Number of Arguments
car	hd	head	1
cdr	tl	tail	1
cons	jn	join	2
atom	at	atom predicate	1
eq	eq	equal predicate	2
quote	'	quote	1
lambda	fnc	function	2
(cond (p x) (t y))	if p x y	if-then-else	3
	\$	no implicit ()'s	1
define	def	define	2
eval	val	value-of	1
	valt	time-limited eval	1
	sexp	m-expr to s-expr	1
	mexp	s-expr to m-expr	1
numberp	#	number predicate	1
plus	+	plus	2
difference	-	minus	2
times	*	times	2
expt	^	raised-to-the	2
quotient	/	divided-by	2
remainder	%	remainder	2
equal	=	equal predicate	2
	!=	not-equal predicate	2
lessp	<	less-than predicate	2
greaterp	>	greater-than predicate	2
	<=	not-greater predicate	2
	>=	not-less predicate	2

Summary of Parenthesis-Free LISP

present a method for getting two new theories of LISP program-size complexity: a theory of program size for the LISP presented in Section 2, which is the subject of this paper, and a theory of parenthesis-free program size for the toy LISP in [3], which we shall say no more about. So we have **four** LISP complexity theories altogether.⁵

It is straightforward to apply to parenthesis-free LISP the techniques I used to study bounded-transfer Turing machines [6–9]. Let us define $H_{\text{pf}}(x)$ where x is a bit string to be the size in characters of the smallest parenthesis-free LISP M-expression whose value is the list x of 0's and 1's. Consider the self-contained defined-from-scratch parenthesis-free version of (append p q):

```
(fnc(app)(app p q)'fnc(x y)if at x y jn hd x(app t1 x y))
|
123456789012345678901234567890123456789012345678901234567
|
10      20      30      40      50
```

Here p is a minimal parenthesis-free LISP M-expression for the bit string x , and q is a minimal parenthesis-free LISP M-expression for the bit string y . I.e., the value of p is the list of bits x and p is $H_{\text{pf}}(x)$ characters long, and the value of q is the list of bits y and q is $H_{\text{pf}}(y)$ characters long. (append p q) evaluates to the concatenation xy of the bit strings x and y and is

$$H_{\text{pf}}(x) + H_{\text{pf}}(y) + 57 - 2$$

characters long. Hence

$$H_{\text{pf}}(xy) \leq H_{\text{pf}}(x) + H_{\text{pf}}(y) + 55.$$

Adding 55 to both sides of this inequality, we have

$$H_{\text{pf}}(xy) + 55 \leq [H_{\text{pf}}(x) + 55] + [H_{\text{pf}}(y) + 55].$$

Therefore, let us define H'_{pf} as follows:

$$H'_{\text{pf}}(x) = H_{\text{pf}}(x) + 55.$$

⁵The next paper in this series [5], on a character-string oriented dialect of LISP, will add one more LISP complexity theory to the list, for a grand total of five!

H'_{pf} is subadditive just like $L(S)$, the maximum bounded-transfer Turing machine state complexity of an n -bit string:

$$H'_{\text{pf}}(xy) \leq H'_{\text{pf}}(x) + H'_{\text{pf}}(y).$$

The discussion of bounded-transfer Turing machines in [6–9] therefore applies practically word for word to H'_{pf} . In particular, let $B(n)$ be the maximum of $H'_{\text{pf}}(s)$ taken over all n -bit strings s :

$$B(n) = \max_{|s|=n} H'_{\text{pf}}(s) = \max_{|s|=n} H_{\text{pf}}(s) + 55.$$

Consider a string s that is $n + m$ bits long and that has the maximum complexity $H'_{\text{pf}}(s)$ possible for an $(n + m)$ -bit string, namely $B(n + m)$. This maximum complexity $(n + m)$ -bit string s can be obtained by concatenating the string u of the first n bits of s with the string v of the last m bits of s . Therefore we have

$$B(n + m) = B(|uv|) = H'_{\text{pf}}(uv) \leq H'_{\text{pf}}(u) + H'_{\text{pf}}(v) \leq B(|u|) + B(|v|) = B(n) + B(m).$$

Thus B is subadditive:

$$B(n + m) \leq B(n) + B(m).$$

This extends to three or more addends. For example:

$$B(n + m + l) \leq B(n + m) + B(l) \leq B(n) + B(m) + B(l).$$

In general, we have:

$$B(n + m + l + \dots) \leq B(n) + B(m) + B(l) + \dots.$$

From this subadditivity property it follows that if we consider an arbitrary n and k :

$$B(n) \leq \left\lfloor \frac{n}{k} \right\rfloor B(k) + \max_{i < k} B(i).$$

Hence

$$B(n) \leq \left(\frac{n}{k} + O(1) \right) B(k) + O(1).$$

[Recall that in this context $O(1)$ denotes a bounded term and $o(1)$ denotes a term that tends to the limit 0.⁶] Dividing through by n and letting $n \rightarrow \infty$, we see that

$$\frac{B(n)}{n} \leq \left(\frac{1}{k} + o(1)\right) B(k) + o(1).$$

[Recall that

$$\begin{aligned} \limsup_{n \rightarrow \infty} \varphi(n) &= \lim_{n \rightarrow \infty} \sup\{\varphi(k) : k \geq n\}, \\ \liminf_{n \rightarrow \infty} \varphi(n) &= \lim_{n \rightarrow \infty} \inf\{\varphi(k) : k \geq n\}. \end{aligned}$$

The **supremum**/**infimum** of a set of reals is the l.u.b./g.l.b. (least upper bound/greatest lower bound) of the set. This extends the **maximum** and **minimum** from finite sets to infinite sets of real numbers.⁷] Therefore

$$\limsup_{n \rightarrow \infty} \frac{B(n)}{n} \leq \frac{B(k)}{k}.$$

Since this holds for any k , it follows that in fact

$$\limsup_{n \rightarrow \infty} \frac{B(n)}{n} = \inf_k \frac{B(k)}{k} = \beta.$$

This shows that as n goes to infinity, $B(n)/n$ tends to the finite limit β from above. Now we shall show that this limit β is greater than zero. It is easy to see that because shorter LISP M-expressions may be extended with blanks,

$$\alpha^{\max_{|s|=n} H_{\text{pf}}(s)} \geq 2^n.$$

Here α is the number of characters in the parenthesis-free LISP alphabet. (This inequality merely states that LISP M-expressions of at least this size are needed to be able to produce all 2^n n -bit strings.⁸) In other

⁶See HARDY and WRIGHT [10, p. 7].

⁷See HARDY [11].

⁸If it weren't for the fact that all shorter expressions are already included in this count, this inequality would have the following slightly more cumbersome form:

$$\sum_{k \leq \max_{|s|=n} H_{\text{pf}}(s)} \alpha^k \geq 2^n.$$

words,

$$2^{(\log_2 \alpha) \max_{|s|=n} H_{\text{pf}}(s)} \geq 2^n.$$

Thus

$$(\log_2 \alpha) \max_{|s|=n} H_{\text{pf}}(s) \geq n.$$

I.e.,

$$\max_{|s|=n} H_{\text{pf}}(s) \geq \frac{n}{\log_2 \alpha}.$$

Hence

$$\frac{\max_{|s|=n} H_{\text{pf}}(s)}{n} \geq \frac{1}{\log_2 \alpha}.$$

Therefore

$$\frac{B(n)}{n} = \frac{\max_{|s|=n} H_{\text{pf}}(s) + 55}{n} \geq \frac{1}{\log_2 \alpha} + \frac{O(1)}{n} = \frac{1}{\log_2 \alpha} + o(1).$$

Thus we see that for all sufficiently large n , $B(n)/n$ is bounded away from zero:

$$0 < \frac{1}{\log_2 \alpha} \leq \liminf_{n \rightarrow \infty} \frac{B(n)}{n}.$$

Hence the finite limit

$$\lim_{n \rightarrow \infty} \frac{B(n)}{n} = \beta,$$

which we already know exists, must be greater than zero. We can thus finally conclude that $B(n)$ is asymptotic from above to a nonzero constant β times n :

$$\begin{aligned} B(n) &\sim \beta n, \\ &\geq \beta n. \end{aligned}$$

I.e., the “adjusted by +55” maximum parenthesis-free LISP complexity $H'_{\text{pf}}(s)$ of an n -bit string s is asymptotic from above to a nonzero constant β times n :

$$\begin{aligned} \max_{|s|=n} H'_{\text{pf}}(s) &\sim \beta n, \\ &\geq \beta n. \end{aligned}$$

In other words, the maximum parenthesis-free LISP complexity $H_{\text{pf}}(s)$ of an n -bit string s is asymptotic to a nonzero constant β times n :

$$\begin{aligned} \max_{|s|=n} H_{\text{pf}}(s) &\sim \beta n, \\ &\geq \beta n - 55. \end{aligned}$$

4. The Halting Probabilities Ω_{pf} , Ω'_{pf} , Ω''_{pf}

In Section 3 we showed that the maximum possible parenthesis-free LISP complexity H_{pf} of an n -bit string is asymptotic to βn , just as was the case with H_{LISP} in [1, Section 8], even though we no longer count all parentheses as part of the complexity of a LISP S-expression. With this basic result in hand, one can immediately rework all of [1] for this new complexity measure H_{pf} .

What about reworking the sequel [4], which studies the corresponding incompleteness theorems and halting probabilities? Everything is straightforward and immediate. The only problems that arise in reworking the discussion in [4] to use H_{pf} instead of H_{LISP} , are with halting probabilities. We must figure out (a) how to define a new halting probability Ω_{pf} to replace Ω_{LISP} , and (b) how to prove that the initial n -bit segment of the new version Ω'_{pf} of Ω'_{LISP} has $H_{\text{pf}} \sim \beta n$.

Ω_{pf}

Problem: How do we define a new halting probability Ω_{pf} to replace Ω_{LISP} ?

As was discussed in [4, Section 4], the sum

$$\Omega_{\text{LISP}} = \sum_{(e) \text{ halts}} \alpha^{-|(e)|}$$

is ≤ 1 and converges because non-atomic LISP S-expressions are self-delimiting. I.e., no extension of an (e) included in Ω_{LISP} is included in Ω_{LISP} . However, extensions of non-atomic parenthesis-free LISP M-expressions may yield other valid M-expressions.

There is a simple general solution to this problem: In our imagination we add a blank at the end of each parenthesis-free LISP M-expression to cut off possible extensions. With this imaginary modification, it is again the case that no extension of a parenthesis-free LISP M-expression is another valid parenthesis-free LISP M-expression, just as was the case with non-atomic S-expressions in [4, Section 4]. In other words, we define the parenthesis-free LISP halting probability as follows:

$$\Omega_{\text{pf}} = \sum_{e \text{ halts}} \alpha^{-|e|-1}.$$

This is summed over all parenthesis-free LISP M-expressions e that have a value or are defined. In [4, Section 4] this sum is taken over all **non-atomic** LISP S-expressions e that have a value or are defined. Here we do not have to restrict the expressions e included in the sum to be non-atomic. We have $\Omega_{\text{pf}} \leq 1$, as desired.

Ω''_{pf}

The exact same method used to produce the halting probability in [4, Section 8], Ω''_{LISP} , immediately yields a corresponding Ω''_{pf} .

$$\Omega''_{\text{pf}} = \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} \alpha_{ij} 2^{-2^{\lceil \log_2 i \rceil - 2^{\lceil \log_2 j \rceil - 2}}.$$

Here

$$\alpha_{ij} = \frac{\# \text{ of } j\text{-bit strings that have parenthesis-free LISP complexity } \leq i}{\# \text{ of } j\text{-bit strings}}.$$

Just as in [4, Section 8], Ω''_{pf} has the property that the string consisting of the first n bits of the base-two expansion of Ω''_{pf} asymptotically has maximum possible parenthesis-free LISP complexity βn . Thus we can follow [4, Section 5] and construct from Ω''_{pf} 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 [4, Section 5] about equation D_1 or the first n cases of the yes/no question Q_2 in [4, Section 5] about equation D_2 requires a formal system with parenthesis-free 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''_{pf} must have parenthesis-free LISP complexity $> \beta n + o(n)$.

Ω'_{pf}

Following [4, Section 7], $\Omega'_{\text{pf}} = \sum_{n=1}^{\infty}$ of

$$\frac{\# \text{ of different } \leq n \text{ character parenthesis-free LISP M-expressions that halt}}{2^{\lceil \log_2(\text{total } \# \text{ of different } \leq n \text{ character parenthesis-free LISP M-expressions}) \rceil + 2^{\lceil \log_2 n \rceil + 1}}.$$

The proof in [4, Section 7] that an initial n -bit segment of Ω'_{LISP} asymptotically has maximum possible complexity βn no longer works for Ω'_{pf} . The problem is showing that there is a real number γ such that

$$\gamma n \sim \log_2 S_n$$

where

$$S_n = \# \text{ of different } \leq n \text{ character parenthesis-free LISP M-expressions.}$$

As was the case in defining Ω_{pf} , the trick is to imagine an extra blank at the end of each M-expression. In other words, everywhere “ $\leq n$ character M-expressions” appears, one must change this to “ $< n$ character M-expressions.” So the fix is that now we consider instead

$$S_n = \# \text{ of different } < n \text{ character parenthesis-free LISP M-expressions.}$$

As was the case with the definition of Ω_{pf} , this trick automatically takes care of the fact that atomic M-expressions are not self-delimiting. For Ω'_{pf} it is not necessary to follow the proof in [4, Section 7] and consider S'_n , which is the number of expressions counted in S_n that are non-atomic. Instead one works directly with S_n , and it is now the case that $S_{nk+3} \geq (S_k)^n$.

Thus Ω'_{pf} has the property that the string consisting of the first n bits of the base-two expansion of Ω'_{pf} asymptotically has maximum possible parenthesis-free LISP complexity βn . Thus we can follow [4, Section 5] and construct from Ω'_{pf} 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 [4, Section 5] about equation D_1 or the first n cases of the yes/no question Q_2 in [4, Section 5] about equation D_2 requires a formal system with parenthesis-free 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'_{pf} must have parenthesis-free LISP complexity $> \beta n + o(n)$.

In summary, we see that all of [1, 4] carries over from LISP complexity to parenthesis-free LISP complexity.

5. Improving Parenthesis-Free LISP

Following [3], here is an improvement to parenthesis-free LISP.

Let's add `let`, read as "let... be...". `let` has three arguments. If the first argument is an atom, the M-expression

```
let x v e
```

stands for the S-expression

```
((fnc (x) e) v)
```

I.e., evaluate `e` with `x` bound to the value of `v`. On the other hand, the M-expression

```
let (f x y...) d e
```

stands for the S-expression

```
((fnc (f) e) ('(fnc (x y...) d)))
```

I.e., evaluate `e` with `f` bound to the definition of a function whose formal parameters are `x y...` and having `d` as the body of its definition.

Here is an example, a **single** M-expression:

```
let (app x y) if at x y
                jn hd x (app tl x y)
let x '(a b c)
let y '(d e f)
(app x y)
```

The value of this expression is:

```
(a b c d e f)
```

Evaluating the following **four** M-expressions gives the same final value, but leaves `app`, `x`, and `y` defined.

```
def (app x y) if at x y
                jn hd x (app tl x y)
def x '(a b c)
def y '(d e f)
(app x y)
```

The `let` notation makes our parenthesis-free LISP more convenient to use, and all the proofs in Sections 3 and 4 go through without change with `let` added.

References

- [1] G. J. CHAITIN, “LISP program-size complexity,” *Applied Mathematics and Computation* **49** (1992), 79–93.
- [2] J. MCCARTHY et al., *LISP 1.5 Programmer’s Manual*, Cambridge MA: MIT Press (1962).
- [3] G. J. CHAITIN, *Algorithmic Information Theory*, 3rd Printing, Cambridge: Cambridge University Press (1990).
- [4] G. J. CHAITIN, “LISP program-size complexity II,” *Applied Mathematics and Computation*, in press.
- [5] G. J. CHAITIN, “LISP program-size complexity IV,” *Applied Mathematics and Computation*, in press.
- [6] G. J. CHAITIN, “On the length of programs for computing finite binary sequences by bounded-transfer Turing machines,” Abstract 66T–26, *AMS Notices* **13** (1966), 133.
- [7] G. J. CHAITIN, “On the length of programs for computing finite binary sequences by bounded-transfer Turing machines II,” Abstract 631–6, *AMS Notices* **13** (1966), 228–229.
- [8] G. J. CHAITIN, “On the length of programs for computing finite binary sequences,” *Journal of the ACM* **13** (1966), 547–569.
- [9] G. J. CHAITIN, “On the length of programs for computing finite binary sequences: Statistical considerations,” *Journal of the ACM* **16** (1969), 145–159.
- [10] G. H. HARDY and E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Oxford: Clarendon Press (1990).
- [11] G. H. HARDY, *A Course of Pure Mathematics*, Cambridge: Cambridge University Press (1952).