A Lisp-Language Mathematica-to-Lisp Translator

Richard J. Fateman
University of California, Berkeley

Abstract

Wide distribution of the computer algebra system Mathematica[1] has encouraged numerous programmers, researchers and educators to produce libraries of programs in its special language, for incorporation as "packages" into Mathematica systems. Although some features of the language are quite interesting, some authors have found that for their purposes the Mathematica proprietary computer program has problematical and difficult-to-alter semantics. Therefore certain kinds of experiments and developments are necessarily inconvenient. An initial step in opening up such user-written libraries to re-use is an independent re-implementation of the language via a non-proprietary parser. In principle, this allows other implementations of semantics, as well as experiments in data representation, while still using the language basically as described in the Mathematica references. We describe a parser written in Common Lisp, a language which is appropriate for three reasons: (1) It is a standard and has wide distribution; (2) It supports numerous useful features including automatic storage allocation and garbage collection, arbitrary-precision integers, and tools for lexical scanning of languages; and (3) Lisp is the host language for several algebraic manipulation systems whose subroutines may be of some interest for programmers implementing alternative semantics.

Introduction

The Mathematica [2] computer algebra system has significantly raised the level of attention devoted to the area of symbol computation. There are groups of investigators writing programs in the Mathematica language for a variety of tasks, ranging from physics calculations to the tutoring of calculus students.

Because such libraries of programs are necessarily in Mathematica's special language, they can be interpreted only with the specific semantics of that proprietary computer program. Contrary to the situation in some experimental languages, the system is a "black box" and it is difficult to alter the semantics in fundamental ways. For example, if one objects to the quality (or perhaps efficiency) of floating-point calculations, there is not much choice available.

By breaking the language away from the system through writing a non-proprietary parser, we allow other implementations of semantics, including but not limited to algorithms for simplification, floating-point arithmetic, integration, and representation.

This report describes a parser for the user-language for Mathematica, written entirely in Common Lisp.

The Program: in Brief

We wrote a Common Lisp program that can read (from a file or a keyboard) virtually any Mathematica program or command, and will produce a Lisp data structure closely resembling the `FullForm` printout of Mathematica. Persons familiar with Lisp who have examined Mathematica's `FullForm` probably will have noticed the close resemblance. We have modified the structure only slightly to make the form more natural in Lisp. The modification is easily stated: The Mathematica expression `x[y]` in `FullForm` is, in Lisp, `(x y)`. For example, the input syntax `a&&b` which has a `FullForm` of `And[a,b]` becomes the Lisp symbolic expression `(And a b)`.

Lexical Analysis and Parsing

After trying (with only modest success) various mostly-automatic parsing techniques, we ended up with a basically hand-coded parser. Mathematica, by contrast with some other computer algebra systems, does not feature an extensible syntax: this suggested that the implementation was, in fact, somewhat *ad hoc*. Our second approach was to write about one page of Lisp "program-writing programs" (macros) to develop the basic lexical analyzer as a dispatch table from a Common Lisp "read-table." We used a macro-expanded version of the procedures (expanded to about five pages) because of various irregularities in the Mathematica

[1] Mathematica is a trademark of Wolfram Research Inc.

syntax which could most easily be addressed by hand-modification of some of the programs, rather than further "macro-ology".

In addition to the five pages of lexical analyzer, there are another ten pages of program, consisting of about 6 utility programs and about 45 other procedures. For each syntactic construction one or two procedures sufficed. The names, such as `parse-plus`, `parse-times`, and `parse-and` correspond to the syntactic structures being parsed. Most of the functions are quite regular in appearance, resembling each other substantially with only a few substitutions for different levels in the precedence hierarchy. Starting with a few templates for left-associative, right-associative, n-ary, and special cases for bracketing constructions, we ended up with a hand-programmed recursive-descent 1-token look-ahead parser.

Given the incomplete and slightly inaccurate description of the grammar ([2] Appendix B), we found it time-consuming to overcome a variety of peculiarities in the language. In particular, there is a somewhat irregular treatment of "white-space" and one must determine, at every `#\newline` (in the Lisp notation) whether the characters collected so far constituted a valid program.

Diagnostics are fairly crude: in case of an error, the parser halts, leaving unscanned the the first token representing the failure to constitute a prefix of a valid sentence; the remainder of the characters beyond this token are still unread.

Although some expressions are parsed which are either not accepted by Mathematica syntactically, or are rejected by Mathematica semantically, these do not seem to represent problems.

Comments on the Language

Mathematica has a large number of tokens, some of which are rather confusing when juxtaposed. Some tokens appear within or as prefixes or suffixes of other tokens. For example, `!b` is used as a prefix notation for `Not[b]`, and `a!` is used as a postfix notation for `Factorial[a]`. The construction `a!!` is "double-factorial" or `Factorial2[a]`. The notation `a b` is `(Times a b)`. Considering these notations, what should the expression `a! !b` mean? Mathematica (as well as our parser) interpretes it as `(Times (Factorial(Factorial a)) b)` although one might prefer another interpretation: `(Times (Factorial a) (Not b))`. Adding to the confusion, if one omits a space, `a!!b` is parsed as `(Times (Factorial2 a) b)`. It is rather non-obvious what happens when one inserts spaces in the construction `a++b`.

The challenge for a symbolic mathematical language is to make the notation for simple and common expressions easy to use, yet to allow for natural extension to a large span of mathematics. Mathematica is somewhat idiosyncratic in the first place, and somewhat irregular in supporting the second. Emphasizing a single underlying data-type (a tree), and associating programming language "hacks" with mathematical operations (`Listable`, `Flat`) may lead to subtle mathematical bugs. Nevertheless, the expressiveness of the language is substantial and it can be used in ways reminiscent of puzzlers in Lisp and (especially) one-liners in APL.

Open-ended features

Presumably a user of this program will wish to write an evaluation scheme for the symbolic expressions produced by the parser. Two "open-ended" areas are the treatment of integers, parsed as (for example) `(Integer 5)` and the real numbers, which are parsed into pairs of the pre- and post- decimal point integers. For example, 1.20 is `(Real 1 20)`. This by no means limits the kind of representation or construction allowed.

Other Modifications

We found some puzzling parses in Mathematica. For example `a<b>c` parses as `Inequality[a, Less, b, Greater, c]` yet `a<b==c` parses as `Equal[Less[a, b], c]`. Even though `Inequality` appears to be undocumented, we prefer to parse the latter as `(Inequality a Less b Equal c)`. This changes the syntax only with respect to more-than-2-argument comparisons: a construction eschewed by most other languages entirely.

We parse `1.2.3` as `(Dot (Real 1 2) 3)` whereas Mathematica version 1.2 gives an error.

The use of Mathematica `Packages` and `Contexts` is not supported (nor is it inhibited) – it need not be treated as a syntactic issue: we believe it can be handled by an interpreter without change to the syntax. Although mapping into Lisp packages is an obvious approach, it would require further study. Merely recreating the implementation of hiding of information in Mathematica may be less than ideal.

Tests

We have parsed various library files in a standard version 1.2 Mathematica distribution, without encountering any syntax errors. Examination of the output (by eye) suggests the translation is accurate. Detailed checks on each construction (some of which are actually

quite unlikely to have much meaning) were also performed, comparing the Mathematica form to the Lisp form.

We timed our program to parse (and generate internal structure for) a library file for piecewise integration of 950 lines and 32,000 characters of Mathematica code. It took about 5.7 CPU seconds on a MIPS M/120 machine and about 26 seconds on a VAX-11/785, each running Allegro Common Lisp. It took about 15.5 CPU seconds on an IBM RT(AP) running Lucid Common Lisp. Default "optimization" settings were used. In each case, about half the CPU time is spend in the collection of character for tokens.

We estimate that about 30% of the time of the parsing time is taken up in checking for the termination of a "sentence" at each "newline" – in our implementation, a costly "convenience" of the Mathematica language.

What Next

Clearly a rudimentary evaluation / simplification / pattern matching system along the lines of Mathematica's should be considered. Experimentation with the semantics would then be simpler. Although much of the semantics would seem to be dictated by even rudimentary correspondences to the Mathematica syntax, there is a substantial area for modification. We have, for example, written a prototype simplification program that may have asymptotically superior complexity to Mathematica's. We believe that the underlying support of Common Lisp is used to good effect in these additional tasks.

Why Lisp

The lexical analyzer and parser use some Lisp features: atoms, property lists, compilation into machine language, read-tables, lists, characters and arbitrary precision integers. The built-in storage allocation features of garbage-collection are naturally used. These features, plus major support for prototyping and debugging, were positive aspects of choosing Lisp.

In principle, a major disadvantage in using Common Lisp as a base is that most implementations require large amounts of memory, typically dwarfing in size this application. Since it is not particularly important to us to make this experimental system run on small computers, this is not an issue.

More significant is the possibility of using, in the same base system, subroutines from Macsyma, Reduce, or Scratchpad, all written in Lisp.

The sole non-standard Common Lisp feature we expect is that a system must provide a way to read programs which use both upper and lower case characters: otherwise our program produces `(PLUS X Y)` instead of `(Plus x y)` for the input `x+y`.

Availability of the Program

A copy of the program (as well as this paper in LaTeX form) is available via `ftp` from an arpanet site. The program is about 944 lines, 28,000 characters. Since an announcement on an electronic bulletin board (sci.math.symbolic) in January, 1990, free copies have been distributed to sites in Norway, France, Sweden, the Netherlands, and 7 sites in the USA. Contact the author (fateman@berkeley.edu) for details on further distribution.

Acknowledgments

This work was supported in part by the following grants: National Science Foundation under grant number CCR-8812843 through the Center for Pure and Applied Mathematics, University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871 under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley; NSF grant CSD-8722788 (ERL); the IBM Corporation and a matching grant from the State of California MICRO program. Students M. DesJardins and P. Klier contributed earlier partial versions of a Mathematica parser.

References

[1] Roman Maeder, *Programming in Mathematica*, Addison-Wesley Publ. Co., Redwood City, CA., 1989.

[2] Stephen Wolfram, *Mathematica – A System for Doing Mathematics by Computer*, Addison-Wesley Publ. Co., Redwood City, CA., 1988.