SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 6 August 2007; **38**:509–522 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

The benefits of posing software as a language interpreter

W. Van Snyder

Jet Populsion Laboratory, Pasadena, CA 91109

SUMMARY

Complicated and comprehensive software that is meant to execute in a non-interactive or semi-interactive mode needs to be configured to carry out the desired tasks, needs to carry out those tasks efficiently, needs to be extensible to take on additional ambitions, and needs to be maintainable. All of these goals can be advanced by posing the software as a language interpreter. Herein, we describe the application of that principle to data analysis software for the Microwave Limb Sounder instrument on the Earth Observing System Aura satellite.

Introduction

Data analysis software for the Microwave Limb Sounder (MLS) instrument on the Earth Observing System Aura satellite [1] is divided into four major programs. The second of these, called "Level 2," or more tersely "L2," is charged with the task of analyzing the microwave emission of the atmosphere (the "radiance"), as observed by the instrument, to deduce the concentration of trace constituents of the atmosphere, and its temperature, at roughly 3,500 profiles extending from ten to one hundred kilometers altitude, each profile containing 37 points, every day.

As one might expect, extensive computations are required, but that is not the end of the story. In addition to the radiance, additional data, including but not limited to a spectroscopy catalog, antenna patterns, filter shapes, orbit and attitude data, and an initial guess for the parameters of interest are required, and that's also not the end of the story. Radiances in some spectral bands are useful to observe one molecule but irrelevant to another, or are useful at one altitude but not another, but that's also not the end of the story. Emission

Contract/grant sponsor: This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with NASA.

from different molecules can most efficiently be modeled by different computational methods. Finally, the same models that are inverted to determine the temperature and composition of the atmosphere can be used, given a sample atmosphere, to produce simulated instrument data sets, and to produce, offline, derivatives of radiance with respect to composition and temperature, for use by one of the methods to deduce composition from radiance.

The details of the foregoing are not really relevant to the remainder of this report. The intent is to convince the reader that numerous factors contribute to the computational organization and progress.

Other than the radiance, orbit and attitude data, and initial guess (which is obtained from climatological averages), all of the considerations arising from the foregoing could be incorporated into the program as initial data and ordinary program decision making. From experience with a previous instrument and its analysis software, we learned that configuring the software to operate efficiently, reliably and accurately requires extensive experimental tuning. Statements in a general-purpose programming language are very low-level representations of the configuration of a program. As such, if the configuration were expressed by ordinary decision-making statements in the program, it would be tedious to change, and more error prone than if it were expressed at a higher level. Perhaps more importantly, since the size of the program now exceeds 200,000 lines, recertification after each tiny tweak of the configuration could be hideously expensive.

Therefore, we chose to configure the MLS programs by input. In a bygone era, that input might have been a sequence of numbers, carefully organized in a rigid sequence. Tuning a configuration expressed in that way would have been nearly as difficult as tuning one expressed within the program. An inch further along, one might have used something less rigid, such as Fortran NAMELIST. Fortunately, we have more computational resources at our disposal, and more software technology upon which to draw.

We instead chose to pose the configuation specification as a "little language," and to organize the program as an interpreter of that language.

Superficial description of compiler technology

To process the input, we use conventional techniques borrowed from compilers. The characters of input are grouped into "tokens" (analogous to words and punctuation marks in natural languages) by a process called "lexing." The structure of sequences of tokens is then recognized by a process called "parsing," which is analogous to grammatical analysis of sentences in natural language. In middle school we are taught to draw "sentence diagrams." The program uses an analogous process, producing a sentence diagram called an "abstract syntax tree" (just "tree" from now on). Just as with natural language, it is possible in synthetic languages to express perfectly grammatical nonsense, so the first step after constructing the tree is a first-cut sanity check. Finally, the program traverses the tree in a depth-first left-to-right order, carrying out computations it recognizes to be demanded thereby.

The first three of those processes are described superficially here. The interested reader is referred to a text specifically concerned with compiler technology for more details [2], [3]. The fourth is then described in more detail.

Syntax

We chose an IDL-like [4] syntax, but others might have different tastes. In outline, each specification consists of a word followed by a list of zero or more name-value pairs, each pair preceded by a comma. The entire specification can be preceded by an optional label followed by a colon.

Lexing

Lexing is carried out by a deterministic finite automaton. Each invocation of the lexer returns one token, which consists of its "part of speech" (name, number, plus sign...), the index of its text in a "string table" (so no further searching is needed when the text is used), and where it appeared in the input (for error reporting).

The string table consists of three data structures. The first is an array of characters that holds the text of the tokens. The second is an array of numbers, the zeroth element of which is zero, and after that the i^{th} one is the index in the character table of the last character of the i^{th} string. If we call the character array C and the string table S, the text of string i is in $C(S(i-1)+1:S(i))$ and the length of the string is $S(i)-S(i-1)$.

Input is read into the end of the character array. As lexing proceeds, if the text of a token is new it is added to the end of the string table and its string index is the index of the newly-created string; otherwise, its string index is the index of the found string.

To facilitate finding whether a string is a new one, the third data structure is a hash table [5].

The module that manages the string table provides numerous inquiry procedures such as `get_string`, `display_string`, `enter_string` and `string_length`.

Parsing

Although it would not be irrational to use an LR parser driven by a table generated by a parser generator such as YACC [6], we chose to carry out parsing using a handwritten recursive-descent (LL) parser, to avoid the need for dependence upon yet another program.

No matter what parsing methodology is chosen, if the result needs to be examined more than once, it's useful to represent the "sentence diagram" by an abstract syntax tree (abstract because extraneous details such as comments have been striped away). In the case of MLS L2, we traverse at least part of the tree once for each "chunk" of data (about 15 profiles). We have chosen to represent the tree using "CDR code" [7].

CDR code represents a tree by writing the sons of a vertex as consecutive array elements. Then, each vertex that has descendants includes the index of its first son and the number of sons. Vertices at the "leaves" of the tree represent names, numbers and strings, by replacing the "first son" index with an index in the string table. The representation is compact, can be easily produced either by LL or LR parsers, and can be referenced quickly, but is difficult to modify. In our application, however, there is no need to modify it.

In addition to procedures used by the parser to build the tree, the module that manages the tree provides procedures to access it, such as `nsons`, `subtree` (actually the index of the root of the subtree), `node_id`, and `sub_rosa` (the string index in a leaf).

Type checking

The syntax allows the value of each field of a specification to be a number, a word, a more general expression, or a string, but if a field is expected to be a numeric expression, the label of another specification, or a string, isn't appropriate (i.e., it's grammatically correct nonsense). One of the benefits of posing software as a language interpreter is that the type of each value can be checked, relations between input items can be specified and checked, and the physical units of numeric inputs can be specified and checked, and converted to standard units (e.g., kilometers or megahertz, even if the inputs are meters or gigahertz), all within a single framework.

The MLS L2 program encodes the requirements for each specification – what fields it can have — and for each field — what types of values are allowed — using a tree. The type-specification tree is built in the same array that holds the abstract syntax tree, before the parser runs. This is done by program statements, not as a result of input, as it doesn't change frequently (but changes are quite simple). After the parser runs, the two trees are joined by a new root node.

Then, a single process traverses the entire tree, aided by a data structure called the "declaration table," which is a two-level structure similar to the string table (thereby allowing several definitions for each word), indexed by the string index. As the tree is traversed declarations of words are entered into the declaration table. Vertices within subtrees related to declarations may be connected one to another using a field in each tree vertex called the "decoration," and ultimately to the declaration table. When a reference to a word is discovered, the tree vertex of the reference is linked to a tree vertex related to the declaration. The structure of the declaration tree is used to determine that the fields of a specification are allowed, and that the types of values of the fields (including the labels of other specifications) are allowed. This unified type checking system relieves the remainder of the program of this responsibility. Units checking for numeric values could have been incorporated into this framework, but the importance of doing so was not recognized at the time of its design; since other development has been of higher priority, it has not yet been so incorporated, so each process that evaluates an expression is responsible for units checking (conversion to standard units is done automatically within the expression evaluator).

Traversing the type-checked tree

After type checking is complete, the actions proper of the program are carried out by a second depth-first left-to-right traversal of the part of the tree that arose from the input (i.e., skipping the type definitions). The program uses the name of each specification (actually an index created by the type-checking process) as a selector to carry out an action. For example, the procedure that processes a `vector` specification allocates space for a vector (and decorates the

tree vertex of the specification with the vector's index in the vector database). Each procedure that processes a specification uses the fields in the specification to complete its definition. Continuing the vector example, a field specifies which quantities the vector contains. When a field for which the value is required to be a vector is processed, the decoration of the field's tree vertex indicates the position in the tree of the vector declaration (thanks to the type checker), whose declaration in turn is the index of the vector as described above. It is clear that the program can be configured to deal with any desired number of vectors, each with any desired collection of quantities (which are in turn defined by declarations within the configuration specification); this flexibility extends to all of the data structures to which the configuration specification has access.

Ultimately, when used to process satellite data, `retrieve` specifications are encountered; the program typically spends 94% of its time evaluating the forward model of the atmosphere, and nearly all of the remaining 6% doing linear algebra related to inverting the model, all of which is triggered by a `retrieve` specification. Thus, it is clear that posing the program as a language interpreter has imposed essentially no measurable cost compared to the major mission of the program.

Additional benefits

In addition to automatic and complete type checking, several other benefits accrue as a consequence of posing the program as a language interpreter. First, of course, is that the type checking is concentrated in one process, so if errors are discovered they can be corrected in only that place. If a similar process is carried out throughout a program, it may be slightly different, and subtly differently incorrect, in each place. Thus concentrating type checking in one place reduces the chance for difficult-to-find subtle errors, thereby increasing reliability and reducing maintence cost.

As new ambitions for the program arise, it has proven to be easy to add them, either by allowing new values to existing fields (usually new literals for enumerated types but occasionally by allowing new types), new fields for existing specifications, or new specifications. In most cases the new facilities were added by minor modifications of existing procedures (especially in the case of new values of existing fields or new fields), or by entirely new procedures that are nearly independent of other procedures. Thus posing the program as a language interpreter has allowed modifications to be implemented at relatively lower cost than would be the case with a different design.

As experience with the program was gained, it became clear, as anticipated, that performance, both in terms of running time and accuracy of results, could be improved by tuning the configuration. The configuration for production runs now approaches 25,000 specifications. Although this tuning has been a time-consuming process, it would have been far more difficult if the configuration specification were more rigid. Posing the program as a language interpreter provides the necessary flexibility in the configuration specification.

Conclusions

Posing a program as a language interpreter confers several benefits. First, it allows checking that values in the input have the correct types and the correct relations one to another. Second, it allows type checking to be concentrated in one place, thereby reducing development and maintenance cost, and increasing reliability. Third, many of the facets of the input can be processed more independently from one another than might be the case with a more rigid input structure. Fourth, it allows modifications to the program to be implemented more independently than might be the case with a different organization, thereby reducing development and maintenance costs. Fifth, it separates configuration of the program from the program proper, thereby separating instead of conflating certification of the program and its configuration, which reduces certification cost substantially. Sixth, it confers considerable flexibility on the organization of the input, and thereby on the operation of the program. Finally, it deploys well-known existing technology that has a well-understood theoretical and mathematical foundation, without imposing a significant performance penalty.

ACKNOWLEDGEMENTS

The author learned how to construct a program as a language interpreter by attending a class on compiler technology, given by Frank DeRemer and Tom Pennello as part of the Summer Institute for Computer Science at the University of Califonia, Santa Cruz, in 1983. The author subsequently used the framework learned in that class, cast in Pascal, Modula-2, Ada and Fortran 95, in many projects prior to EOS MLS, and to teach undergraduate-level compiler classes for fourteen years. The Fortran 95 instance of the framework, similar to what is used in EOS MLS, accompanies this manuscript. The module `parser_m` is not the LL parser used for MLS L2; rather, it is an interpreter of tables produced by the LR parser generator written by Charles Wetherell and Alfred Shannon [8], which is not included.

REFERENCES

1. Joe W. Waters et. al. The Earth Observing System Microwave Limb Sounder (EOS MLS) on the Aura satellite. *IEEE Transactions on Geoscience and Remote Sensing Special Issue on the EOS Aura Mission*, 44(5), May 2006.
2. Alfred. V. Aho, Ravi Sethi, and Jeffrey. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.
3. Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler with C.* Benjamin/Cummings, New York, 1991.
4. Research Systems, Inc., Boulder, Colorado, USA. *IDL Reference Guide*, April 1998.
5. Donald. E. Knuth. *The Art of Computer Programming: Sorting and Searching.* Addison–Wesley, Reading, Massachusetts, 1973.
6. S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.
7. John McCarthy and Michael Levin, et. al. *LISP 1.5 Programmer's Manual.* MIT, 1966.
8. Charles Wetherell and Alfred Shannon. LR — Automatic parser generator and LR(1) parser. *IEEE Trans. Software Eng.*, 7(3):274–278, 1981.