# Programming with Grammars: an Exercise in Methodology-Directed Language Design

**Eric C. R. Hehner**
Computer Systems Research Group, University of Toronto, Toronto, Canada M5S 1A4
**Brad A. Silverberg**
Apple Computer Corp., 10260 Bandley Drive, Cupertino, CA 95014, USA

**Our premise is that programming languages should be designed to facilitate a programming methodology. We begin with the methodology of Michael Jackson; the notation that results is similar to Hoare's communicating sequential processes. Data and processing structures are described together in one grammatical formalism.**

## INTRODUCTION

It is easy, and all too common in Computing Science, to invent a tool without providing methods for its use. Hardware designers are recognized culprits: though their machines are commonly used via a layer of software that provides the user with a high-level language, it is still rare for their designs to be determined by this use. We programming language people are pleased to point this out,[1–3] but we commit the same sin at our higher level. After the invention of ALGOL 60, we waited ten years to begin to enunciate methods for programming in ALGOL-like languages—so-called 'structured programming'. Our habit is still to design languages or features, then to search for axioms, and finally to propose methods of use based on the axioms. An outstanding exception is Dijkstra's guarded-command notation,[4] which was determined by its methodology.

We begin with the methodology of Michael Jackson,[5] and develop a notation that is intended to correspond to, or facilitate, this methodology. The resulting notation is similar to Hoare's notation for communicating sequential processes.[6]

We illustrate the methodology and the notation with an example: the Telegram Problem.[7,8] The problem is, of course, irrelevant to the point of the paper; it was chosen because it is sufficiently complicated to illustrate the value of the methodology, but no more complicated.

## GRAMMAR AS SPECIFICATION

Problems are usually stated in English (or other natural language), which makes precision difficult. Here is our example problem, omitting one bit of detail to be added later.

> A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits, and blanks . . . . The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters in length are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

As with any informal specification, there is the danger that one person's understanding of the problem may differ from another's. It is not clear whether the input may contain only the empty telegram. It is not clear whether the program should terminate on a telegram with no words, no words except 'ZZZZ', or no chargeable words (from Henderson's solution we may infer that he intended the last meaning). The value of a formal specification to make one's understanding clear and unambiguous has been argued forcefully many times (e.g. Ref. 9). An appropriate way to specify formally the legal inputs to a program is by a grammar; Noonan[10] and McKeeman[11] have done exactly that for the Telegram Problem. The following grammar is adapted from McKeeman's; we shall explain the notation afterward.

```
telestream:  *space; *(telegram; +space);
             "ZZZZ"; (space | eof).
telegram:  +(word; +space); "ZZZZ".
word:  +(letter | digit).        {but not "ZZZZ"}
letter:  'A' | 'B' | . . . | 'Z'.
digit:  '0' | '1' | . . . | '9'.
space:  ' '.
eof:  ⊣.
```

We have made several increasingly common changes to BNF, and some odd changes whose purpose will become clear by the end of the paper. We have dropped the angle brackets from the non-terminals, and at the same time put quotes around the terminals (single quotes for single characters, and double quotes for strings); the advantages are the ability to distinguish terminals from metasymbols, and the ability to specify a blank space. The end-of-file mark ⊣ is acting as a terminal here, but is special in a way that will become clear later. BNF's ':: =' has become ':'. On the right sides we allow any regular expression operators, rather than just concatenation and alternation, and we allow these expressions to be nested, with the aid

CCC–0010–4620/83/0026–0277$02.50

of (meta)parentheses. Alternation is denoted by '|' as usual. Concatenation is denoted by ';'. The unary prefix '*' operator means 'zero or more of', and the unary prefix '+' means 'one or more of'. Other operators, such as '?' meaning 'optional', are possible. Each grammar rule ends with a period.

Following "ZZZZ" in *telestream*, a space or end-of-file mark is needed so that the stream is not terminated by a word beginning "ZZZZ". The same role is played by the two occurrences of *+space*. Whether this grammar corresponds to Henderson's intent is not important; it makes clear the understanding that we shall adopt. The need for a comment in the rule for *word* points out an inadequacy of context-free grammars. Though we could specify all finite sequences of letters and digits except "ZZZZ" with context-free (even regular) rules, it is awkward to do so. McKeeman introduces conjunction and difference operators that solve the immediate problem; however, there are sometimes constraints that require a more powerful formalism. We shall allow the insertion of Boolean expressions into the grammar rules. In the example, we want to insert the expression *word* $\neq$ "ZZZZ" at an appropriate place. This notation is derived from a particular form of two-level grammar called 'affix grammar'.[12,13]

We have stated that a grammar is a tool for the precise specification of the possible inputs to a program. Often, a programmer is not told what the input looks like; it is part of his task to decide that. Every such programmer is a language designer; the set of possible inputs is a language. The design will be better (simpler, more elegant, more complete) if the language designers' best tools are used.

## GRAMMAR AS MODEL

A grammar does more than just specify a set of strings: it also specifies a structure. There are many grammars that describe the same set of legal inputs, but a particular grammar shows a particular way of structuring the problem that can serve as a model for the solution. Jackson, Noonan and McKeeman[5,10,11] present solutions to the Telegram Problem, in COBOL, SIMPL-T and PL/I, respectively, using grammars as models. Jackson proposes that programs should be designed this way. His method produces, in essence, an LL(1) parser for those grammars that are LL(1); for those that are not, some backtracking is introduced.

For some problems, a single grammar is insufficient as a model for the entire program. The input grammar can always serve as a model for the portion of the program that reads the input, but other grammars may be needed for other parts. We shall return to this point later; for the present, we shall assume that the structure of the input is the structure of the entire solution.

## GRAMMAR AS PROGRAM

A grammar is not just a model for a program that reads input: it *is* a program that reads input. Using a parser generator as compiler, one need not translate to a more conventional programming language. A parser generator can supply an algorithm, such as LALR(1), that is more powerful than one a programmer can reasonably write, such as LL(1). Thus more input structures can be accommodated with less effort. A parser generator can perform optimizations that are too complicated or tedious to be done by hand. If a better parsing algorithm is discovered, programs written as grammars take immediate advantage, and the programmer need not know the details of the new algorithm. The current practice is that each programmer writes for each program what is, in essence, an *ad hoc* parsing algorithm to recognize the input. We propose to free the programmer from having to supply any particular parsing algorithm; he writes only a descriptive grammar.

Our point would be complete if recognizing the input were a separate activity, but usually one needs to embed further processing and output into the grammar/program. Compiler writers have called this 'attaching semantics to syntax', and have invented special notations for the purpose.[12–14] These notations have come far toward making grammars enhanced with semantic annotations a practical compiler-writing tool. The remaining awkwardness can be removed, we claim, by thinking of grammars as programs in a programming language, that is, by applying principles that have been successful in programming language design. For example, grammar symbols in an enhanced grammar may have inherited and synthesized attributes[14] (or inherited and derived affixes[12]). These are similar to input and output parameters in programming languages. If we maintain separate terminologies and forms, we may miss the relevance of the considerable literature about parameter forms and mechanisms.

## NOTATION

The following is not intended to be a complete nor a definitive description of a language, nor is the partially described language intended to be exemplary. We present just enough notation to allow us to present our example of the programming method.

(1) The body (right side) of a grammar rule may contain variable declarations, assignments, Boolean expressions, input and output (see (4) below). The scope of variables is local to the rule in which they are declared.

(2) Non-terminals may have parameters. For example, the rule

*jack* (*x*:**integer**): body.

declares that *x* is an integer-valued input parameter within the body of this rule. When *jack* is used (invoked) within the body of some rule, an integer-valued expression must be supplied as argument.

(3) Non-terminals may return results. For example, the rule

*george* (**result** *y*:**integer**): body.

declares that *george* may be used in the body of a rule where an integer value is required. Its value will be the final value of variable *y*.

(4) Instead of terminals, which are an implicit input, we shall have explicit input effected by

*input* → variable

We shall assume that this assigns the next character

of input to the variable until finally it assigns the special value ⊣ denoting end-of-file. Output is effected by

*output*←expression

We shall assume that the value of the expression is printed appropriately.

(5) Some data types are:
   (a) **integer**—as usual
   (b) **boolean**—**true** and **false**
   (c) **character**—e.g. 'A'
   (d) **string**—e.g. "ABC"

For integer *i*, character *c*, and string *s*,

" " denotes the null string
*s*||*c* denotes the extension of *s* by *c*
*s*[*i*] denotes the *i*th character of *s*
#*s* denotes the length of *s*.

## EXAMPLE OF METHOD

The programming method consists of two steps: write a grammar, then embed processing in it. In this section we illustrate the method, in the notation we have just given, on our example problem. A grammar describing the input is:

```
telestream: *telegram; endword.
telegram: +teleword; endword.
teleword: word ≠ "ZZZZ".
endword: word = "ZZZZ".
word (result w:string):
    var c:character;
    w := " ";
    *(input→c; c = ' ');
    +(input→c; c ≠ ' '; c ≠ ⊣ ; w := w||c);
    input→c; (c = ' '|c = ⊣).
```

Only the rules for *telestream* and *telegram* require embedded processing.

```
telestream: output←"TELEGRAM ANALYSIS";
            *telegram; endword;
            output←"END OF ANALYSIS".
telegram: var wc, lwc: integer, w: string;
          wc := 0; {word count}
          lwc := {longword count}
          +(w := teleword;
              (w ≠ "STOP"; wc := wc + 1;
                           (# w ≤ 12
                           |# w > 12;
                            lwc := lwc + 1)
              |w = "STOP");
            output←w; output←" ");
          endword;
          output←"WORD COUNT:";
          output←wc;
          output←"LONG WORD COUNT:";
          output←lwc.
```

Here we have used *teleword* as a string, so we must revise its rule slightly.

```
teleword (result w:string): w := word; w ≠ "ZZZZ".
```

The other rules remain unchanged.

Though we began with a grammar describing the input, we could equally well have begun with a grammar describing the output. In this example, the input and output structures are compatible in that they can be described together in one grammar. We now discuss what to do when the input and output structures are incompatible.

## COMMUNICATING GRAMMARS

As stated earlier, some problems require several grammars, one for each part of a solution. We shall illustrate this shortly, but first we give a notation by which grammars can communicate. In fact, we have already used the notation, calling it input/output. A grammar receives information (from another grammar) by the notation

source→variable

where source is a local name, i.e. an identifier whose meaning is local to the grammar in which it appears, that identifies a source of information. A grammar sends information (to another grammar) by the notation

destination←expression

where destination is a local name. (In our example, *input* and *output* are local identifiers within the *telestream* grammar that identify a source and a destination of information. We have not yet hooked this grammar to the outside world.)

A program is a collection of grammars together with a communication graph. A communication graph is a set of declarations, each of which connects a destination of one grammar with a source of another. We use the start/goal non-terminal of a grammar to identify the grammar. For example, the declaration

*G*1·*out*→*G*2·*in*

means that, within grammar *G*1, the statement

*out*←1

may be used to send the value 1 to grammar *G*2, within which it may be received by the statement

*in*→*v*

and assigned to variable *v*. A grammar may have several sources and several destinations for information. When a grammar completes its execution, the special value ⊣ is sent from it to all its destinations.

When considered as an undirected graph, whose nodes are grammars and whose doubly-labelled edges represent communication (ignoring the direction of communication), a communication graph must contain no cycles. This constraint is easily checked by a compiler. It is necessary (a weaker constraint cannot be stated independent of the internal structure of the grammar) and sufficient to guarantee that queues (buffers) are not required and that deadlock is not possible.

When only one processor is available for execution, the grammars act as co-routines, with the arrow notation meaning 'resume'. When there are as many processors as grammars, the grammars may be executed concurrently with the arrow meaning communication, and providing synchronization. When the number of processors available is properly between 1 and the number of grammars,

the implementation may be partly co-routine and partly concurrent execution.

## EXAMPLE OF METHOD, RESUMED

Our initial statement of the Telegram Problem had one incomplete sentence so that our initial example would require only a single grammar. We now complete the sentence to give an example that requires two grammars.

> This stream is available as a sequence of letters, digits, and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed.

We shall assume that the device is a card reader, which produces the next card (80-character string) of input, until finally it produces the end of file mark $\dashv$, just as a grammar does. There is no structural relationship between the telegrams and words on the one hand, and the cards on the other; the former may be wholly within a card, partly on one card and partly on the next, or even spanning several cards.

A grammar describing the stream of cards is easily written.

*cardstream*: **var** *card*: **string**;
*(*reader*→*card*; *card* ≠ ⊣);
*reader*→*card*; *card* = ⊣.

For any two structures, there is always a common substructure (if nothing else, the bit). For communication between grammars, we usually choose the largest common substructure. In our example, both words and cards are composed of characters, so that will be the unit of communication from the *cardstream* to the *telestream* grammar. Embedding the processing in our new grammar is now easy.

*cardstream*: **var** *card*: **string**, *i*: **integer**;
*( *reader*→*card*; *card* ≠ ⊣ ; *i* := 1;
+( *i* ≤ 80; *out*←*card*[*i*]; *i* := *i* + 1);
*i* = 81);
*reader*→*card*; *card* = ⊣.

The communication graph contains

*cardstream*·*out*→*telestream*·*input*

plus two other declarations, one to link the card reader to *cardstream*, and the other to link *telestream* to the printer.

If the intention of the problem is that a word cannot continue across a card boundary, then a card boundary acts as a blank. We need only insert *out*←' ' after the expression $i = 81$. In this case communication could have been via the larger unit *word*.

## CLASS OF GRAMMARS

In conventional programming languages, an alternative construct (**if** . . . **then** . . . **else**, . . ., or **case**) is LL(1) in the sense that the decision must be made at the beginning of the alternatives, on the basis of one expression, which alternative to execute. Dijkstra's **if** guarded-command-set **fi** construct[4] is non-deterministic in the sense that if more than one guard is true, then any one of the corresponding commands may be executed—each one must be sufficient to establish the desired result. It is still, however, LL(1) in our sense. The word 'non-deterministic' is used more commonly (and less aptly) in parsing theory to describe a grammar in which a particular alternative may have to be chosen (the choice is not a free one), but the choice cannot be made at the beginning of the alternatives, or even at their end, or for an unbounded distance following them. The usual single-processor implementation is called 'backtracking'. The alternative construct of our notation allows non-determinism in this sense.

There are classes of grammars between LL(1) and context-free for which efficient parsers can be built. Ken Day[15] has designed an LR parser generator that accepts those programs that are LALR(1) grammars, and produces instructions (actually, tables) in which backtracking is never required. The decision among several alternatives can be determined by their first point of difference, or even by the first expression following the alternatives.

In our telegram solution, the number of repetitions of an iterative construct is sometimes determined by a following expression. For example,

$+(i \le 80; S); i = 81$

when written recursively, has the form

$R; i = 81$

where

$R: (i \le 80; S \mid i \le 80; S; R).$

The choice between the alternatives of $R$ is determined by the expression $i = 81$ which follows $R$. That construct is therefore LR(1).

## CONCLUSION

The analogy between grammatical formalisms and programming languages, between grammars and programs, between parser generators and compilers, is complete. By thinking of grammars as programs, we apply principles of language design that improve our grammatical notations; and our programming languages improve by using the structures developed for grammars.

The language we have (partly) described, based on a context-free grammatical formalism that allows regular-expression right sides with embedded predicates and assignments, is higher-level than current programming languages in the following sense. A programmer describes the structure of his problem, i.e. the data structures and associated processing, in one notation. He does not describe data and processing structures separately, with different notations for fundamentally similar structures. He does not supply the uninteresting detail of the algorithms to recognize the input data, the algorithm to control and synchronize the processing, nor the algorithm to generate output.

For a computation-oriented problem (with not much data), a solution in our language is much like a solution in a conventional programming language. The language

distinguishes itself in more data-oriented problems, where the grammatical basis can be used to advantage. There, our solutions specify the data structures clearly and concisely. This follows from the programming methodology we have chosen. And that is the main point: the methodology does not follow from the language design; rather, the language is an embodiment of a methodology.

**Acknowledgement**

We gratefully acknowledge the critical reading of an early draft by David Elliott.

## REFERENCES

1. W. M. McKeeman, Language directed computer design. *Proc. AFIPS 1967 FJCC*, Vol. 31, AFIPS Press, Montvale, N.J., pp. 413–417 (1967).
2. D. B. Wortman, A study of language directed machine design. *Ph.D. Thesis*, Stanford University, Palo Alto (1972).
3. E. C. R. Hehner, Computer design to minimize memory requirements. *Computer* **9** (8), 65–70 (1976).
4. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, New Jersey (1976).
5. M. A. Jackson, *Principles of Program Design*, Academic Press (1975).
6. C. A. R. Hoare, Communicating sequential processes. *Comm. ACM* **21**(8), pp. 666–677 (1978).
7. P. Henderson and R. Snowden, An experiment in structured programming. *BIT* **12**, 38–53 (1972).
8. C. B. Jones, *Software Development: a Rigorous Approach*, Prentice-Hall International, London (1980).
9. D. L. Parnas, The use of precise specifications in the development of software. *1977 IFIP Congress Proceedings*, Vol. 7, North Holland (1977).
10. R. E. Noonan, Structured programming and formal specification. *IEEE Trans Software Eng.* **SE-1** (4), 421–425 (1975).
11. W. M. McKeeman, Respecifying the telegram problem. *Technical Report*, University of California at Santa Cruz (1977).
12. C. H. A. Koster, Affix Grammars in *ALGOL 68 Implementation*, edited by J. E. Peck, North Holland (1974).
13. D. A. Watt, The parsing problem for affix grammars. *Acta Informatica* **8**, 1–20 (1977).
14. D. E. Knuth, Semantics of context-free languages. *Mathematical Systems Theory* **2**, 127–145 (1968).
15. K. R. Day, Alegra: a language for expressing grammars and algorithms. *M.Sc. Thesis*, Department of Computer Science, University of Toronto (1982).

Received November 1982