Design of a geophysical programming language

Ronald Ullmann

Introduction

In the middle 1950's, Fortran was introduced as a language for implementing algebraic formulas on the computer. Fortran can convert mathematical formulas that are understandable by the programmer into programs that the computer could execute. However, Fortran had the disadvantage of operating on one number at a time, and the programmer has to program the input and output of each number. In the middle 1960's, APL was introduced as a new notation for formulas and algorithms. APL was compact and could operate on whole vectors in one equation. However, the input and output features of APL from data files is not very simple, and APL programs are hard to understand because of APL's compact notation. There is no programming language around that suits the needs of geophysicists who work on seismic data. I think that geophysicists have four basic requirements for a programming language:

1. The language should offer many of functions that are commonly used in geophysical functions. Some of these functions are Fourier transforms, data smoothing operations, and digital filters. The interpreter of the language should be smart enough so that when the user calls a function, the interpreter will determine the parameters needed to execute the function, such as the length of the vector.
2. The language should be able to work on whole traces at a time. The programmer should have the ability to add, subtract, or Fourier transform whole traces without having to specify the iterations for each element of the trace.
3. The language should have simple input and output control that uses the trace as the basic I/O unit. The programmer should have access to traces from different data sets and have the ability to send any intermediate and final results to different files.

4. The language should be simple for the programmer to use and for others to understand. Suppose the programmer wants to read a trace, take the Fourier transform, smooth the spectrum, write the amplitude spectrum of the smoothed spectrum to one file, inverse Fourier transform the smoothed spectrum, and write the results to another file. The instructions for this series operation should be both simple and explicit.

I have noticed the lack of such a language recently when I began working on different deconvolution formulas that operate on whole traces at a time. Usually, these formulas will differ from each other by a minor change. For example, one formula may be

$$result = \mathbf{F}^{-1}\left[\left(\frac{\ll Y\ \bar{Y}\gg}{<Y\ \bar{Y}>}\right)^{1/2} Y\right],$$

where Y is the Fourier transform of the trace, $\mathbf{F}^{-1}$ is the inverse Fourier transform, and the operator $< >$ is a smoothing operator. The new formula could be

$$result = \mathbf{F}^{-1}\left[\left(<\frac{<Y\ \bar{Y}>}{Y\ \bar{Y}}>\right)^{1/2} Y\right].$$

In order to try the new formula, I rewrote parts of my program that computed the first formula and then recompiled it. If I wanted to try the first formula on different data, I would rewrite the program again. Each time I wanted to try a different formula, I had the risk of not changing the program correctly and thus getting erroneous results. Once, the erroneous results looked very good, but I could never figure out what I did to duplicate it. One solution was to save a copy of a program for each formula. Unfortunately, the programs would take up too much memory. If I just recorded the changes, then I would have a hard time backtracking if I wanted to use a previous formula. Therefore, I decided to write a intelligent program that would read in and interpret the formulas that I supplied. With this smarter program, I only had to record the formulas, and I was freed from compiling the program after a little change in a formula. I also had the extra benefit of spending less time debugging my program. I only had to check the input formula.

In order to write this program, I had to define a language to communicate the formulas and some rules for how the computer interprets the formulas. I am calling the language Atp. The purpose of this article is to define the syntax for Atp and the rules of how the formulas should be interpreted. For the rest of this article, I will refer to traces as vectors, since they are both arrays of numbers. I will also refer to Atp as both a language and a program that implements this language.

prog ::= *block* | < *range* > *block* { < *range* > *block* }

range ::= *subrange* { , *range* }

subrange ::= *rangenum* { : *rangenum* }

rangenum ::= *integer* | **$** | **(** *rangeexpr* **)**

rangeexpr ::= *rangenum* { **+** *rangenum* | **-** *rangenum* }

block ::= *equa*; { *equa*; }

equa ::= **&***outcmd* **=** *expr* | *term* **=** *expr*

expr ::= *mult-term* { **+** *mult-term* | **-** *mult-term* }

mult-term ::= *factor* { ***** *factor* | **/** *factor* }

factor ::= *term* | *constant* | **[***complexnum***]** | **(***expr***)** | *func* | **&***incmd*

func ::= **fft (***arg***)** | **ift (***arg***)** | **sm (***arg***)** | **ssm (***arg***)** | **ccj (***arg***)** |
sqr (*arg***)** | **inv (***arg***)**

arg ::= *expr* { , *expr* }

term ::= **A** | **B** | ... | **Z**

incmd ::= *inc* | *inc* **[** *file* **]** | *inc* **[** *file* **;** *offset* **]**

inc ::= **cin** | **iin** | **rin**

outcmd ::= *outc* | *outc* **[** *file* **]**

outc ::= **aout** | **cout** | **iout** | **pout** | **rout**

file ::= **0** | **1** | ... | **9**

offset ::= *integer* | **-** *integer*

constant ::= *number*

complexnum ::= *number*, *number*

TABLE 1. Above is the syntax for Atp. The symbols that are printed in bold-face type are terminal symbols that make up the program. The italic words are strings of terminal symbols. The characters "::=", "|", "{", and "}" are symbols that control the construction of the various statements. The symbol *number* refers to any valid floating point number, and the symbol *integer* refers to any valid fixed point integer.

Definition and implementation of Atp

Table 1 above shows the complete production rules, or syntax, of Atp in Backus-Naur form (Naur, 1963). The bold words and characters are terminal symbols, which are the symbols used by the programmer to construct the program. The italic words are non-terminal symbols and represent sequences of terminal words. The symbol "::=" means "is defined as". Any non-terminal symbol on the left side of "::=" can be replaced by the expression on the right side. The symbol "|" is an alternative operator. This operator separates

alternatives available to the programmer in replacing non-terminal symbols. For example, a factor can either be a term, a constant, a complex number, an expression, a function, or an input command. Anything within the braces, "{" and "}", can be repeated any number of times. For example, a block may contain any number of equations, as long as each equation ends with a semicolon.

Every valid statement in Atp can be formed using the production rules listed in Table 1. By starting at the production rule for *prog* and replacing non-terminal symbols, any Atp program can be formed. The production rules are used by a parser to determine whether a Atp program has a valid syntax. A parser is a program that determines whether a program written in some language follows the production rules of the language. If the program violates a production rule, then the program has a syntax error. If the programming language is an LL(1) language, which roughly stands for one-symbol-lookahead without backtracking, than a simple parser can be written. A LL(1) language requires that a parser only look at the next symbol. The parser should not have to backtrack in order to determine which alternative is present. Atp almost meets the conditions for a LL(1) language. Atp fails because in order to determine whether a minus sign in front of a number is the subtraction operator or a negative operator, the lexical analyzer has examine the symbols before the minus sign. In every other respect, Atp is a LL(1) language. Horowitz (1983) explains the conditions that a language must meet for it to be a LL(1) language.

A LL(1) language can be parsed using a top-down parsing algorithm. For example, the equation

```
B = fft ( A * B + C );
```

can be formed by using the production rules for the non-terminal symbols *equa*, *expr*, *multterm*, *factor*, *func*, and *term*. Figure 1 shows the tree structure formed when this equation is parsed. Wirth (1976) describes a basic procedure of converting from syntax rules to a working parser. First, the production rules are converted into flow graphs. An example of a flow graph is shown in Figure 2, which is the flow graph for the non-terminal symbol *expr*. The parser starts at the left hand side and travels to the right, choosing its path based on the next symbol it finds. In this case, the parser looks for a *multterm* right away. The parser then checks to see if the next symbol is either a "+" or a "-". If it is, the parser looks for another *multterm*. The parser will keep looping as long as it detects either a "+" or a "-" after each *multterm*. If next symbol is something else, than the parser goes on to the next stage. This flow graph can be translated directly into a Pascal procedure, which is shown in Figure 3. The parser for a Atp program can be written in any high level language using this approach. However, the parser is much easier to write and maintain in a language

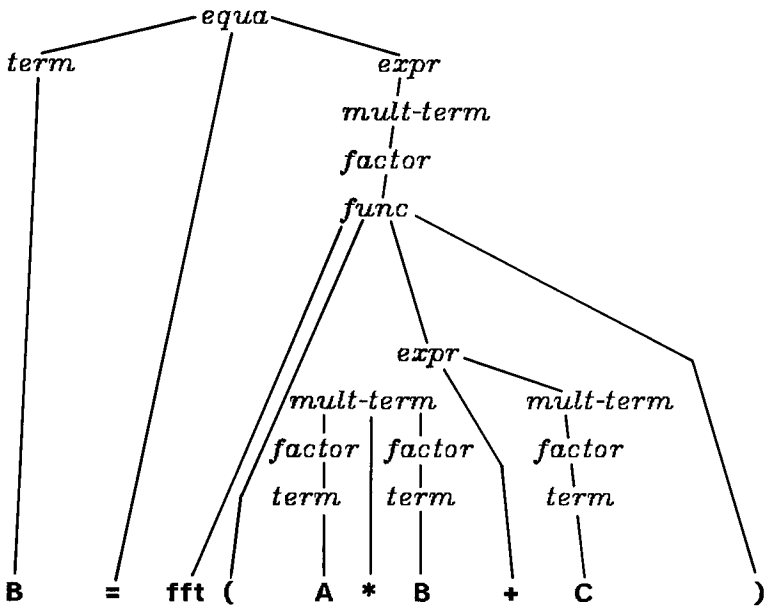

FIG. 1. Parsing diagram for a simple equation in Atp.

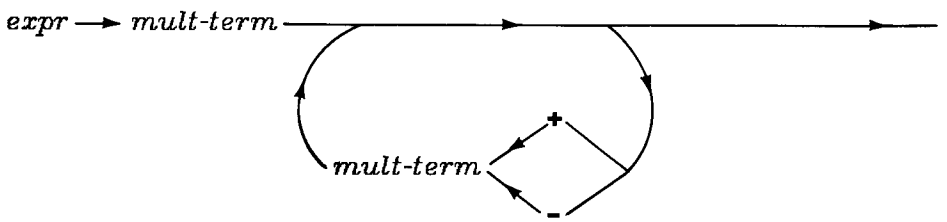

FIG. 2. A flow graph for the production rule for *expr*. The program always moves from the left to the right in tracing the graph.

that allows recursion, such as Pascal or C. In a recursive language, each non-terminal symbol can represent a call to a procedure. In Figure 3, *multterm* would be a procedure that checks the next set of symbols in the input for factors.

A program which implements Atp has three basic parts, which are shown in Figure 4. The first part is a lexical analyzer. The lexical analyzer reads the equations entered by the user and converts them to a form that can be handled more easily by the parser. This means removing tabs and blanks, removing comments, converting numbers to an internal format, and recognizing the functions, I/O commands, and the variables. The analyzer also has the responsibility of detecting illegal characters and checking that the number of parenthesis

```
procedure expr;
  begin
    multterm;
    while nextsymbol in ['*', '/'] do begin
      multterm
    end
  end;
```

FIG. 3. The translated Pascal program of the flow graph in Figure 2. The command *multterm* is a call to a procedure that examines the next symbols for a *mult-term*. The command *nextsymbol* is a function that returns the next symbol in the program.

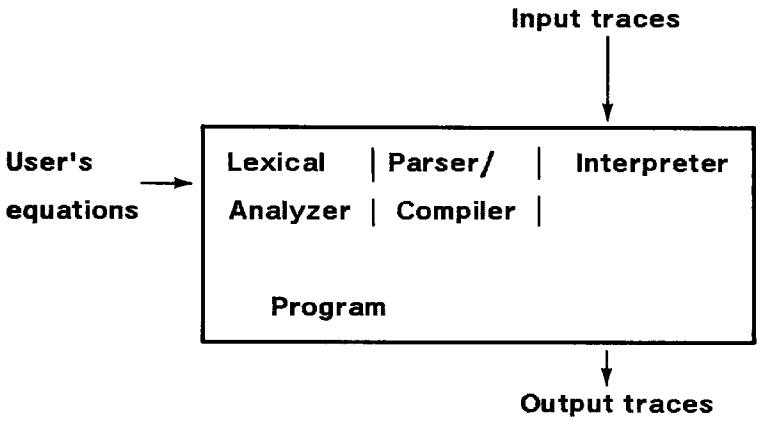

FIG. 4. This diagram shows how a program that implements Atp goes through three stages.

match. The second part of a program combines the parser, which was described above, with a compiler. A Atp program is compiled at the same time it is parsed. A list of commands are formed by the parser and stored in memory. When the parser is done, control is passed to the last part of the program, which is called the interpretor. The interpretor executes the commands provided in the command list. The interpretor handles all the input, output, and operations of the traces.

```
(1)     A = C + &rin[1];      " This is a comment
(2)     B = ift (sm (fft (A)));
(3)     &rout[1] = A - B;     " &rout[1]= B - A
(4)     C = (A + B)/2.;
```

FIG. 5. This figure demonstrates a simple Atp program. The four statements above form a block. For each iteration, the statements in the block are repeated.

Using Atp

Input Format. The input format for a program refers to the position certain items must have. For example, the labels for a Fortran program are confined to the first six columns of each line, and any continuation of a line is marked by a non-blank character in the seventh column of the next line. The input format for a Atp program has no such restrictions. A statement can start anywhere on the line and can extend over several lines. The only restriction is that each statement ends with a semicolon and ranges end with the symbol >. The programmer can use tabs and spaces in any manner to help make the program more readable. The lexical analyzer for Atp treats spaces, tabs, and new lines as white space and removes them from the input.

Blocks and Statements. The basic unit of a Atp program is a block, which consist of a series of statements. Consider the simple Atp program in Figure 5. Each of the four lines are statements, and the set of four lines is a block. Note that statements are separated by semicolons, which means that more than one statement can go on a line. The program that implements Atp repeats the statements in the block for a fixed number of iterations that is determined at the start. By default, the number of iterations is equal to the number of traces there are coming from the input. Later, I will discuss a method for extending the number of iterations. The program that implements the block above, it will read in the first trace and then executes the statements in the block, using the first trace as data. After all of the statements in the block have been executed, the program will go back and execute the statements again, but this time with the second trace. Atp will execute the contents of the block for every trace that is read in. Later on, I will discuss how the programmer can use different blocks on different traces.

Comments. Figure 5 shows the format for comments. Any text that occurs after double quotes, ", on a line is a comment and will be ignored by the parser. The extra equation on line 3 will be ignored by the parser since it occurs after the double quotes. The contents of line 4 are still included in the program since the comment on line 3 ends at the carriage

return. A well written program in any language includes enough comments so that others can read understand the purpose of the program.

Variables. Vectors are manipulated in Atp through the use of variable names, which are the capital letters **A-Z.** Traces are manipulated the same way single numbers are manipulated by variable names in Fortran. The fourth statement in Figure 5 adds the elements of the vector represented by **A** to the elements of vector **B**, divides the sum by 2, and stores the results in vector **C.** During the execution of this statement, the contents of **A** and **B** do not change. If **C** was also on the right side of the equal sign, then the contents of **C** would not change until the right hand side is evaluated. In other words, the right hand side of an equation is evaluated before the contents are assigned to the left hand side. One the first iteration of a block, all the elements in all the vectors are set to zero. For example, the first statement in Figure 5 is adding **C** to the input. The contents of **C** are zero, so the addition is an identity operation. When Atp executes the block for the second trace, then the contents of **C** will be left over from what was assigned to it in statement four. In other words, Atp does not reinitialize the contents of a variable after each iteration. If there was no assignment statement for **C** in the first iteration, then the values of **C** will remain zero.

Vector Length. For the purposes of internal calculation, Atp assumes that all of the vectors used in calculations have the same number of elements or length. Atp takes the length of the input vector supplied by the user and finds the next highest power of two. For example, if all the input vectors each have 1000 numbers, than Atp uses vectors that have a length of 1024. The input vectors are padded with zeros when they are read in and stored in memory. This conversion is done in order that the fast Fourier transform may be used. All of the output vectors from Atp have the power of two length. Atp also assumes that all of the vectors consist of complex numbers. However, Atp can convert real numbers to complex using input and output commands that will be discussed later.

Operators and Scalar Numbers. The basic binary mathematical operations, addition, subtraction, multiplication, and division, are defined for the vectors. The symbols used for these operations is the same as in Fortran. The hierarchy for these operations is shown in Table 2. In addition, scalar numbers can also be one of the operands of a binary operation. For example,

```
5 + A
```

adds a 5 to every element of **A**, while

```
[0., 1.] * B
```

multiplies every element of **B** by $\sqrt{-1} = i$. In the fourth statement of Figure 5, all of the

1	"( )", "[ ]"
2	*functions calls*
3	"*", "/"
4	"+", "-"
5	"="
6	":"
7	","

TABLE 2. This table shows the order of evaluation of the different operators available to Atp. The operators on line 1 are executed first, then the operators on line 2, and so on.

elements of the sum of **A** and **B** are divided by 2 before they are transferred to **C.** Complex numbers are written in a Atp program as two numbers enclosed in brackets and separated by a comma. If the number is not enclosed in brackets, the Atp assumes the number is real. The format of floating point numbers is the same as Fortran. For example, $0.05e-4$ becomes $5.x10^{-6}$. Any binary operation involving a vector and a scalar number is identical to a binary operation between the vector and a vector containing the scalar number. The result of a binary operation between two scalars is a vector containing the result of the operation. For example, the statement

```
A = 5 + 7;
```

places a 12 in every entry of the vector **A.**

Name	**Operation**
fft	forward Fourier transform
ift	inverse Fourier transform
sm	smoothing through convolution
ssm	super-smoothing through convolution
ccj	complex conjugate
sqr	square root
inv	reciprocal

TABLE 3. This table lists the functions that are currently defined for Atp.

Functions. I have included certain functions in Atp that are commonly used in geophysical processing. The names of these functions and what they do are listed in Table 3. A function is defined to as an operation that uses one or more arguments to calculate a new vector. Some examples of of how functions are used in Atp are shown in the second line of Figure 5. First, the name of the function is given, followed by a list of the arguments enclosed by a pair of parenthesis. When a function has more than one argument, then arguments are separated in the list by commas. In the current version of Atp, there are no defined functions that need more than one argument. An argument to a function can be a expression, a term, a scalar number, or a vector from the input file. Since all functions are call-by-value, none of the arguments are changed by the function. In other words, functions only use the values of the arguments without changing the arguments. Other functions will be added to Atp in the future, such as the Hilbert transform and high pass filters. One of the properties of Atp is that anyone can define a new function and add it to the language. For example, if a user needs a bandpass filter with specific cutoffs, the user can write the appropriate subroutine and plug it into Atp. Since Atp is currently written in C, any one at the SEP can write their functions in C or Fortran and add them to the current list of functions. The format for new functions has been kept simple to encourage other programmers to write new functions.

Name	**Input/Output Operation**
aout	write the amplitudes of the traces
cout	write the complex form of the traces
iout	write the imaginary part of the traces
pout	write the phase of the traces
rout	write the real part of the traces
cin	read the traces in complex form
iin	read the traces as imaginary numbers and convert it to complex
rin	read the traces as real numbers and convert it to complex

TABLE 4. This table lists the Input/Output commands that are currently defined for Atp.

Input and Output. The input and output commands of Atp are listed in Table 4. Since all vectors in Atp consist of complex numbers, there are a variety of input and output commands to match what the programmer wants to work in. For example, if the programmer wants to read input data that consists of real numbers, than he would use the command **rin.** This command reads the real numbers of a vector into a complex array and assigns zeros to the imaginary part. If the programmer wanted to write only the imaginary part of the result, he would use the command **iout,** which would write only the imaginary part of the vector. All input and output commands are preceded by a **&** to distinguish the commands from variables and function names. This means that the user can use the input and output commands in the same way that he would use variables. Whenever Atp encounters an input command in a statement, it reads in the next vector, places the new vector in a temporary variable, and substitutes the temporary variable for the input command. Likewise, whenever Atp encounters an output command, it substitutes the output command with a temporary variable. When the temporary variable has been assigned its value, its contents are written. The output commands can only be placed on the left side of the equal sign because results are sent to the file. Similarly, input commands can only be placed on the right side of the equal sign because vectors are coming from it.

Each input and output command is followed by a descriptor of which file is being used. The descriptor consists of an integer from 0 to 9 inclosed by brackets, "[ ]". Before Atp is used, the user provides a two lists of files that he will use for input and output. One list will contain the input files, and the second list will contain the output files. The integer in the file descriptor specifies the location of the file name in the list. For example, the equation

```
A = B + &rin[3];
```

reads real numbers from the input file that is third in the input file list, stores the numbers in a complex array, adds numbers to the contents of **B,** and places the results in **A.** If the equation were

```
&aout[1] = fft(&rin[2]);
```

Atp would read in the real numbers from the second input file, find the Fourier transform, and write the amplitude spectrum to the first output file. If the user does not specify a file descriptor, then the default file is zero for both the input and output commands. File zero at the SEP is reserved for writing and reading data from pipes, since the user cannot specify a file zero in the input and output file lists. This default allows Atp to communicate with other processes that are running at the same time.

Each time an output command is called, the vector is appended to the output file file. This means that all output operations are done sequentially into the files. By default, the same applies to input commands. The next vector is read sequentially from the input file each time one of the input commands is used. There is a mechanism for retrieving vectors out of order, which is done by specifying an offset. The offset is a positive or negative integer that is placed after a semicolon and the file descriptor within the brackets. When the offset is not specified, then offset is set to zero. Whenever a trace is read in, an internal pointer in Atp is moved to the next trace in the input file. The value of the offset integer indicates which vector to read relative to this internal pointer. For example, the statement

```
A = &rin[2; -1];
```

means that the previous vector from file 2 should be read in and assigned to variable **A.** The internal input pointer moves to the next trace only when the offset is zero. This means that the statement above has no effect on the position of the pointer after the input operation because its offset is not zero. A positive offset of n means that the input vector should come nth vector after the input pointer. The input commands of the program in Figure 6 show how offsets can be used. Atp will return with an error if the user tries to use an offset that points to a location where there is not data. For example, if the user tries an offset of -1 before any trace has been read in, the user will be trying to read vector zero, which does not exist. A non-zero offset is also not allowed for file 0 since Atp can not back up to read previous data on a pipe. All input from a pipe must be done sequentially.

Range. Suppose the programmer wanted use a certain formula on the first few iterations, a different formula on the middle iterations, and another formula on the last few iterations. The programmer can use the range feature of Atp to specify the block should be executed for any iteration. The range is an positive integer that serves as a type of label for matching a block with a specific iteration. All iterations start at one and go up. The best way to understand how ranges work is to consider the the sample program in Figure 6. The ranges are enclosed in the "< >" symbols on lines 1, 6, and 11. These lines mark the beginning and ending of three different blocks which the user wants to apply to different iterations. The range in line 1 specifies that the statements on lines 2 through 5 are used only for the first iteration. The symbol "$" on line 6 is a macro for the number of the last vector in the data set. If the data set has 50 vectors, then the "$" represents 50. The symbol ":" in line 6 reads as "through". The symbol "-" in line 6 reads as "minus". Therefore, line 6 means that the set of statements on lines 7 through 10 apply for all the iterations from two to the next to forty-nine. Atp requires that all mathematical operations in a range specification take place within parenthesis. Addition and subtraction are currently the only

```
(1)   < 1 >
(2)    A = fft (&rin[1]);
(3)    B = A;
(4)    C = fft (&rin[1]);
(5)    &rout[1] = (A + B + C)/3.;
(6)   < 2 : ($ - 1) >
(7)    A = B;
(8)    B = C;
(9)    C = fft (&rin[1]);
(10)   &rout[1] = (A + B + C)/3.;
(11)  < $ >
(12)   A = B;
(13)   B = C;
(14)   &rout[1] = (A + B + C)/3.;
```

FIG. 6. This is a sample program that shows how ranges work. The numbers on the left hand side are not part of the program; they are for identification purposes.

operations allowed in range expressions. The range in line 11 specifies that the last three statements be used for the 50th iteration of the program. The effect of the program in Figure 6 is to average the spectrum of each input vectors with the spectrum of immediate neighbors and write out the resulting spectrum.

If the result of any range expression is less than one, than Atp returns with an error. However, the programmer can have ranges greater than the number of vectors in the input file. However, the programmer should be careful not to try to read in vectors after reading the last vector. The last block in Figure 6 does not show any input because the last vector was read in by the previous trace. If the user had another block whose range was

```
< ($ + 1) >,
```

then the user could specify some other operation for that iteration, as long as there is no attempt at input. Atp will keep the maximum number it calculates for a range and use it as the number of iterations for the whole program. Blocks can be placed in any order in the program and can be applied at any iteration. If one block was applied to even iterations and another block applied to odd iterations, then the ranges would be

< 1, 3, 5, 7, 9, 11, ... >
BLOCK
< 2, 4, 6, 8, 10, 12, ... >
BLOCK

Figure 7 shows a program that performs the same operations as the program in Figure 6, but uses offsets instead. Note that the program in Figure 7 has more function calls, so it would take longer to run then the program in Figure 6.

```
(1)   < 1 >
(2)    A = fft (&rin[1; 0]);
(3)    B = A;
(4)    C = fft (&rin[1; 1]);
(5)    &rout[1] = (A + B + C)/3.;
(6)   < 2 : ($ - 1) >
(7)    A = fft (&rin[1; -1]);
(8)    B = fft (&rin[1; 0]);
(9)    C = fft (&rin[1; 1]);
(10)   &rout[1] = (A + B + C)/3.;
(11)  < $ >
(12)   A = fft (&rin[1; -1]);
(13)   B = fft (&rin[1; 0]);
(14)   &rout[1] = (A + B + C)/3.;
```

FIG. 6. This is a sample program that duplicates what the program in Figure 6 does. Note that there are more input calls and calls on the function *fft*, which means that this program would take longer to run.

Programming language criteria

There are different criteria for a programming language and different methods of meeting the criteria. The first criteria is that the language should have a well-defined syntactic and semantic description. A well-defined syntax allows the language compiler to recognize ill-formed statements. The well-defined semantic description insures that there is no ambiguous interpretation of any statement. In other words, every statement means only one thing; no other interpretation of the statement is possible. Atp meets this criteria because it

is well defined and any statement means only one thing. A second criteria is that the language can be implemented quickly. In this case, the translation from the programmer's formulas to the command list is very fast and takes up a very minor part of the total run time. Most of the computer time is spent in doing the calculations.

A third criteria is that the language is machine independent. None of the features of the Atp programming language depends on what type system the program runs on. The actual implementation of the commands, such as the input and output, is machine dependent. A program that implements Atp can be written in any high level language, such as Pascal or PL/1. The version that I wrote is in C, which means that any system that has a standard C compiler can use the program I wrote. I did not use any features of the SEP's C compiler that should be available on other standard compilers.

The fourth criteria is that the language should produce efficient code. The efficiency of Atp is limited to the language it is written in and the routines that execute the functions. If these languages make full use of the computer system, then the resulting Atp program should be very efficient. Naturally, a custom written program in Fortran for a particular formula will take less time to run and use less memory than a Atp program that does the same thing. The same principle applies to a program written in assembly language versus one written in Fortran. However, most programmers would rather develop a program in Fortran than in assembly language. Atp is designed primarily as a method of trying many different formulas on some data quickly without having to make major alterations to a program. Once a formula has been found that works, then the programmer should write the custom program.

The last criteria that I use is the flexibility of the language. So far, I have only defined a few functions that I have needed the most. It is possible for others to write subroutines that can be included in Atp. I also have plans to add more abilities to the language, such as variable length vectors, two dimensional arrays, scalar arithmetic, longer variable names, and some type of explicit looping and branching within blocks.

REFERENCES

Horowitz, Ellis, 1983, Fundamentals of programming languages: Rockville, Maryland, Computer Science Press.

Naur, P. ed, 1963, "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, vol. 6, pp. 1-17.

Wirth, Niklaus, 1976, Algorithms + data structures = programs: Englewood Cliffs, New Jersey, Prentice-Hall.