**VECT: an automatic visual Perl programming tool for nonprogrammers**

Hui-Hsien Chou

*BioTechniques 38:615-621 (April 2005)*

Modern high-throughput biological research produces enormous amount of data that must be processed by computers, but many biologists dealing with these data are not professional programmers. Despite increased awareness of interdisciplinary training in bioinformatics, many biologists still find it difficult to create their own computational solutions. VECT, the Visual Extraction and Conversion Tool, has been developed to assist nonprogrammers to create simple bioinformatics without having to master a programming language. VECT provides a unified graphical user interface for data extraction, data conversion, output composition, and Perl code generation. Programming using VECT is achieved by visually performing the desired data extraction, conversion, and output composition tasks using some sample user data. These tasks are then compiled by VECT into an executable Perl program, which can be saved for later use and can carry out the same computation independently of VECT. VECT is released under the GNU General Public License and is freely available for all major computing platforms including Macintosh® OS X, Linux, and Microsoft® Windows® at www.complex.iastate.edu.

**INTRODUCTION**

In the genomics and postgenomics eras, biologists frequently need to process a lot of biological data. Usually, biologists know how their data can be manually handled, but only a few of them are well versed in computer science to be able to turn that into executable code. Powerful bioinformatic tools have been created to solve truly difficult and well-defined problems in computational biology. However, not all needs of biologists are as generic. Actually, most of the time biologists need some bridging programs to connect existing bioinformatic tools together to form their data processing pipeline. These generally involve data extraction, conversion, and reporting tasks that are very specific to their ongoing research. Creating these bridging programs is easy by experienced programmers, but to nonprogrammers, this work can be detrimental and slow.

The author believes this limiting factor of modern biological research can be resolved in a creative manner. In this paper, a visual programming tool, VECT (the Visual Extraction and Conversion Tool), is introduced. It allows users to manipulate their sample data inside its user interface, and then generates Perl programs to replicate these tasks (1). Anyone who needs to process textual format data can potentially benefit from using VECT.

**MATERIALS AND METHODS**

VECT employs a data flow programming paradigm that is different from the control flow programming paradigm more familiar to programmers. An example problem of extracting the translated protein sequences of predicted open reading frames from a GenBank® report (www.ncbi.nlm.nih.gov) is used to illustrate the difference between the two programming paradigms. Suppose both the names and sequences of the proteins must be extracted. These data are delimited by the /protein_id and /translation= tags embedded inside the CDS regions in the report. To extract them, a programmer might have followed the control flow logic shown in Figure 1. A main loop is scanning through all input lines. Each input line is then checked against the name and protein delimiter tags. For the name of a protein, its quoted string name must be extracted. For each protein sequence, things are a bit more complex since it can span several input lines, so an inner loop is run to collect all its parts until the end of the sequence is seen. Subsequently, all quotation marks must be removed, and only when both the name and the sequence of a protein have been collected will output be produced in the FASTA format (www.ncbi.nlm.nih.gov/blast/html/search.html). The scanning continues until all input lines have been seen.

There is nothing wrong with the control flow programming paradigm. In fact, most programmers take it for granted. However, the data flow programming paradigm shown in Figure 1 seems to be an easier approach for nonprogrammers to follow. In this paradigm, focus is placed on how input data can be extracted and processed, disregarding the order of their arrival. For example, obtaining protein names and sequences are considered as two unrelated processes. A user simply needs to define the steps to extract and process them separately (e.g., protein names have to be taken out of quotes, and protein sequences have to be concatenated and then also taken out of quotes). Output is produced using an
output template. Therefore a user does not need to worry whether the name or the sequence of a protein reaches the output template first; the user only needs to know that when they have both arrived, they will be output together using the template as defined.

Although data flow programming used to refer to specialized hardware and software that have never been in widespread use (2), in \textit{V\textsc{ect}}, this is simply the programming method adopted to facilitate user programming effort. The data flow programming paradigm naturally leads to an example-driven programming style. In \textit{V\textsc{ect}}, programming is achieved by letting users handle some sample data in its interface. This is similar to using an editor or a spreadsheet program. \textit{V\textsc{ect}} then translates user actions into executable Perl code expressed in the control flow programming paradigm.

Example-driven programming started in the 1960s and 1970s with the RPG (3) and COBOL (4) programming languages that allowed programmers to format reports using output templates. In the 1980s, spreadsheet programs (5,6) were invented that allowed users to program number-crunching jobs by inserting formulas in some cells and copying them to the others. In the 1990s, the popularity of GUI programs ignited the development of several object-oriented interface libraries and their associated rapid application development (RAD) tools (7,8). These tools allow a user interface to be designed graphically, almost like drawing in a graphic program. The design is then converted into program code that can recreate the interface at runtime. To sum up, programming by examples is not new, but to apply this concept to the data extraction, conversion, and formatting needs of biologists is new to the best of our knowledge. In the following example, the author demonstrates how this form of programming in \textit{V\textsc{ect}} can help biologists create Perl programs.

\textbf{RESULTS}

\textbf{\textit{V\textsc{ect}}} Programming Tutorial

It is helpful to demonstrate how \textit{V\textsc{ect}} works from a user’s perspective before each part of it is explained in detail. The protein extraction problem mentioned earlier is used as an example again. To begin with, a GenBank file is loaded directly into the Input Data panel of \textit{V\textsc{ect}}. The first thing to grab is the protein sequence, so we use the right mouse button to click and drag over \texttt{/translation=}” to set it as an opening block tag. We also set the ending double quote ” as the closing block tag. This defines regions in the input file where data can be selected. Since the ending double quote is not in the same position for each protein sequences, we change both the opening and closing tags to position independent. This allows all protein sequences in the input to be identified and selected. The result is shown in Figure 2A, where pink regions are not selectable, green- and red-colored texts are the opening and closing block tags, respectively, and grey regions are the text actually selected.

The selected data are sent to the Convert Data panel by clicking the Move button. The initial data set is named Protein Parts in the Convert Data panel. Here, additional rules can be added to convert the data. Specifically, we need to add a concatenation rule to connect the broken protein parts into complete protein sequences, and then a quoted data extraction rule is needed to remove the \texttt{/translation=}” and ” tags that are not part of the proteins. The results are shown in Figure 2B. The resulted data set Pure Proteins can be copied to the Output Data panel by clicking the Copy button. In Figure 2C, similar extraction and tag removal steps are defined for protein names. The resulted data set is named Pure Names. Note that yellow-colored texts in this figure indicate line-based selection tags.

To produce the desired output, both Pure Names and Pure Proteins are copied to the Output Data panel. This panel has both Template and Output views. To compose a correct FASTA output, we need to add a greater than symbol $>$ in front of the Pure Names and separate it from Pure Proteins by a new line (see Figure 2D). This FASTA-required symbol is not to be confused with the pair of arrows enclosing data set names. The former is a static text to output, but the latter is a placeholder for data sets. This formatting is taken by \textit{V\textsc{ect}} as a template to group each pair of data from the two sets to produce the output. The results can be checked in the Output view shown in Figure 2E. If they are correct, we can finally go to the Perl Program panel shown in Figure 2F and click the Compile button to obtain a Perl program that can reproduce the same operations. This Perl program can be saved for later use and can work on the other GenBank files with similar contents.

\textbf{Data Extraction}

The Input Data panel of \textit{V\textsc{ect}} allows users to define the extraction of useful data from input files. It is designed to handle semi-structured text files commonly produced by online databases. Selection can be based on fields, in which each field is a sequence of nonwhite characters separated by characters such as tabs or spaces. To select an entire field, just click on the field. Selected data are always highlighted with a grey background color. Selection can also be based on positions relative to either a field or an entire line. By clicking and dragging over a range of characters, a position-based selection is made. If the selected characters are completely contained within a field, then the position selection is relative to the same field on each line. Otherwise, or if the shift key is pressed while dragging, the position selection is relative to the entire line.

Selections can be restricted by designated tags in the input. Tags do not select data per se, but they help define the desired data that are to be selected. Tags can be block opening, block closing, or simply line tags. A line tag allows only lines containing it to be selected. The opening and closing tags enclose a region for selection, but they do not have to be paired. If an opening tag is followed by another opening tag, the second tag defines a new selection region (i.e., it functions both as a closing tag for the previous region and as an opening tag for its own region). All tags are defined by using the right mouse button to select text, but otherwise they are selected exactly the same way as regular text (i.e., tags can also be field- or position-based),
The Control Flow Paradigm

The Data Flow Paradigm

Figure 1. Comparison of the control flow and data flow programming paradigms. Vect (the Visual Extraction and Conversion Tool) presents to its users a data flow programming paradigm shown in the right. Users can separately define how data sources can be extracted, converted, and composed to produce the output. Vect then compiles the design into a Perl program that is expressed in the control flow paradigm shown in the left, which actually implements the computation.
Figure 2. Vecr (the Visual Extraction and Conversion Tool) programming: a tutorial. (A) Each protein block is defined by a green opening tag and a red closing tag. Pink regions are not selectable. The actual selected text is shown in grey. (B) Selected protein fragments are sent to the Convert Data panel and named Protein Parts. Two rules are added to concatenate the fragments (Quoted Proteins) and remove the quotes (Pure Proteins). (C) Similar selection is conducted for protein names. Here the yellow line selection tags are used. (D) The Output Data panel provides a template view to compose user output. (E) The Output Data panel also provides an output view to show the actual output. (F) Finally, a working Perl program can be obtained by clicking the Compile button in the Perl Program panel.
On a Macintosh®, where there is no right mouse button, control-click is used instead. Each time a right-mouse selection is made, a menu will pop up to let users define which kind of tag is being defined. For example, in the tutorial, \( /\text{translation}=\) is defined to be the opening tag, and only by seeing it will a new selection region begin. That is, the content and position of a tag definition are both significant. Blocks can be nested up to a level of seven to accommodate complex selections.

All types of tags can also be made position-independent (i.e., only the content of a tag is used to identify it in input lines). Additionally, block tags can be set to exclusively occupy a line. VECT always indicates nonselectable regions with a pink background, so users can easily experiment with their selection strategy. Once a selection is complete, it can be sent to the Convert Data panel, where data conversions are made. Note that items in a selection always maintain their individuality exactly as they are extracted from the input. For example, in the tutorial, the Protein Parts set is a collection of protein fragments making up the protein sequences. Data individuality is maintained throughout the extraction, conversion, and reporting processes in VECT, unless users convert them into new sets.

Data Conversion

Extracted data usually need to go through some conversion steps before they are in the right format for output. For example, fragments in the Protein Parts set have to be connected to obtain full-length sequences. The Concatenate rule does just that. Concatenation can be based on the level of extraction (i.e., the level of the nested block from where the data are extracted), or it can be based on a delimiter in the extracted data stream to indicate the end of the current concatenation. Individuality is also maintained in the converted set, so the Quoted Protein set contains individual protein sequences joined by each region of labeled \( /\text{translation}=\) and \( " \) tags.

Each time a new rule is added to the Convert Data panel, a new data set is created. The original data set(s) that the new rule is based on will not be altered. This prevents user confusion and allows different conversions to be made from the same source data set. Although each data set is represented separately in VECT, it does not imply that all data sets will be explicitly stored into variables in the generated Perl programs. Doing that is both a memory waste and a performance hit. Usually, data are propagated through the data conversion pipelines, flowing from a source data set all the way to the output, if they can be sequentially converted. VECT stores intermediate data into Perl variables only when data sets with different granularity are being produced or when data from different data sets are being combined asynchronously. The granularity of data sets will be explained later.

Rules provided in VECT are documented online at its distribution web site (www.complex.iastate.edu). Their detail is beyond the page limit of this article. However, three special rules are worth mentioning here. The Select rule operates exclusively in the Input Data panel, and each user-defined selection is instantiated into a Select rule. The Print rule operates exclusively in the Output Data panel, and the output template is translated into a Print rule. These rules make up the sources and sink of all data flowing through VECT and VECT-generated Perl programs. Since there can never be enough predefined rules to deal with all possible user computing needs, the Simple User rule allows users to define their own conversions by writing some Perl code. Because the Perl interpreter is embedded in VECT (see below), VECT can execute the user code and produce the conversion results defined by users. This greatly enhances the flexibility and usefulness of VECT, because writing short Perl code for a missing conversion is much easier than writing a complete Perl program from scratch. Additionally, VECT provides interactive feedback to help users quickly identify any error in their code and resolve it.

Output Composition

The Output Data panel in VECT has two separate views. The Template view is similar to a text editor in which users can type in static text that will be printed as is. Users can also define data set placeholders that are mixed with the static text. A placeholder is delimited by an arrow pair enclosing the name of a data set. The template is used to format items from each of the output data sets. For example, in the tutorial, the symbol > is inserted in front of the Pure Names set to prefix each protein name in the output, which produces a standard FASTA header line. Note that the template is used to format each group of items from the output sets. That means items from output sets are interleaving each other. Because of this fact, if there are different numbers of items in the output sets, VECT will complain because it may be an error (e.g., the protein names and sequences do not match up). Further more, the line width of each output item can be limited (e.g., each protein sequence can be broken into multiple lines of 60 characters).

The concept of output data interleaving takes us to the topic of data set granularity. A VECT-generated Perl program produces output during the preamble, scanning, and finishing stages. During the preamble and finishing stages, the program simply prints any user-supplied prefix or appendix text that is not associated with any output data set (e.g., author name, date, etc.), so there is no granularity issue. Outputs generated during the scanning stage can be associated with three different types of granularities corresponding to line-, region-, and whole input-based data sets. Line- and region-based data sets can be incrementally output during the scanning stage, when an appropriate line is read or a completed region has been seen, but a whole input-based data set cannot be output until the entire input has been scanned. The question is if two data sets of different granularities are composed in the output template, how will they be handled?

There is a closure relationship among the three different granularities when data sets are combined to form new sets or to produce the output. For example, the protein fragments of the Protein Parts set are line-based and can be output when lines containing them are scanned. However, after the concatenation step, the Quoted Proteins set becomes region-based, and each of its protein sequences
can be output only when all fragments of it have been collected. This implies that the Quoted Proteins set must store its temporary results in a Perl variable, so that the concatenated sequence can be output later. In another example, the Quoted Names set is a line-based set. After it is stripped off the quotation marks, the resulted Pure Names set is still line-based and can be output when each item is produced. To sum up, certain rules like the Concatenate rule can produce data sets that have a coarser granularity than its input sets, and any combination of data sets with different granularities will produce a new set having the coarsest granularity among its constituent sets. Therefore, when Pure Names is combined with Pure Proteins, their output becomes a region-based one. Both Vect and Vect-generated Perl code will handle the synchronization necessary to make this automatic granularity promotion work.

**Perf Program Generation**

**Vect** can automatically generate Perl programs, because each of its data extraction, conversion, and output formatting rules has a piece of Perl code attached. Through a user's point-and-click actions, Vect not only displays the rule sets in its various panels, it also records user settings and rule-set interdependency. Therefore, it only takes a compilation process to translate the user work into a Perl program. It is Vect compiler's job to translate user programming done in the data flow paradigm back to the control flow paradigm, which is actually implemented by the generated Perl program.

The granularity and interdependency of data sets determine the order in which they will be collected, converted, and output. Using two different strategies, Vect infers the output that a generated Perl program will produce when it is executed by the Perl interpreter. For efficiency reasons, some rules are directly simulated by Vect. For example, the Select rule, which works entirely in the Input Data panel, is interactively controlled by Vect, so its output must be derived by Vect. Since the Perl interpreter is embedded in Vect (9), other rules (e.g., Pattern and Simple User rules) are handled directly by the Perl interpreter, and their results are stored by Vect. Embedding Perl interpreter inside Vect is achieved through dynamic linking for better compatibility (i.e., the Perl interpreter is not compiled into the Vect executable, but is referred to by Vect during runtime). This requires that Perl be installed on all supported platforms in order to run Vect, but it tolerates the minor difference between the version of Perl used to build Vect and the version that was installed on user machines. Vect supports Perl 5.8 and later. On all UNIX-flavored operating systems and the Macintosh OS X, Perl is bundled by the system vendors. Microsoft® Windows® users can download the Active Perl distribution to install Perl (www.activperl.org). Vect source code is also available for interested users to port it to the other platforms and for older versions of Perl.

**DISCUSSION**

A freely available visual Perl programming tool, Vect, has been created for biologists. It can handle most filter-type data processing jobs similar to the UNIX awk utility (10). The most difficult part of a filter program is to correctly extract the desired data. Even for experienced programmers, some “tweaking” of the data extraction code is often needed before it can function correctly on all input. Once the extracted data are “clean,” it is relatively easier to convert and output them. Vect provides the powerful Input Data panel that interactively displays which data are being selected from where in the input, thus saving users a lot of time in getting their extraction job done right. This is probably the most important feature Vect offers. Users are encouraged to follow the Vect online tutorials to learn how to use it effectively.

Great care has been given to ensure that Vect-generated Perl programs follow good Perl programming styles. The strict and warnings Perl pragma are always enabled, and only “textbook” Perl coding styles are used. Therefore, Vect-generated Perl programs serve both as good learning material and starting points for further refinement. Because the Perl interpreter is embedded in Vect, it can be beneficial to take advantage of the vast amount of Perl modules and scripts that have already been developed for bioinformatics. A future plan is to make accessible to Vect users Perl modules from the Bioperl project (www.bioperl.org) or other similar sources, so Vect users can use advanced Perl modules just like direct Perl programmers can do.

**ACKNOWLEDGMENTS**

The author thanks ISU Complex Computation Lab staff Denise Mooney, Sunyoung Park, Kazuya Suzuki, Satish Venula, and Kent Weber for sample collection, graphics design, and beta-testing, and the authors of the Perl community and the wxWidgets library for making their work freely available for inclusion in Vect. The development of Vect is supported by National Institutes of Health grant no. 4R33GM066400.

**COMPETING INTERESTS STATEMENT**

The author declares no competing interests.

**REFERENCES**

10. Aho, A.V., B.W. Kernighan, and P.J. Wein-

Received 12 August 2004; accepted 1 October 2004.

Address correspondence to:
Hui-Hsien Chou
Departments of Genetics, Development, and Cell Biology and Computer Science
503 Science II Building
Iowa State University
Ames, IA 50011, USA
e-mail: hhchou@iastate.edu