String Pattern Matching in a Visual Programming Language

DAVID JACKSON AND MICHAEL A. BELL*
University of Liverpool†

Abstract

Determining whether or not a pattern of characters is present within a body of text is such a fundamental problem that it has led to a number of notations for the specification of patterns. These include regular expressions, and notations found within string processing and other application-specific languages. Too often, these approaches are either overly simplistic, offering little in the way of pattern matching capabilities, or else they are extremely cryptic and terse.

In the context of visual programming, the pattern matching problem is largely unaddressed; yet the benefits ascribed to such languages offer the potential for extremely powerful and intuitively meaningful pattern notations. In the CALVIN language, designed for the creation of courseware, the authors have used purely visual constructs throughout. In particular, the content of strings may be analysed by using a graphical and augmented version of regular expressions to specify patterns. This notation, and its realization within CALVIN, is described in some detail, and comparisons are made with the more conventional textual form.

Categories and Subject Descriptors: D1.7 [**Programming Techniques**] Visual Programming; D2.6 [**Software Engineering**] Programming Environments — *interactive*; D3.3 [**Programming Languages**] Language Constructs and Features; H1.2 [**Models and Principles**] User/Machine Systems — *human factors*

General Terms: Human Factors, Languages

Additional Key Words and Phrases: String pattern matching, regular expressions, notation

1 Introduction

The problem of searching for a pattern of characters that may or may not be present within a corpus of text is a common computing task, and its satisfactory and efficient solution has

*Michael Bell is supported in his research by the Engineering and Physical Sciences Research Council of Great Britain.
†Authors' address: Department of Computer Science, University of Liverpool, P.O. Box 147, Liverpool L69 3BX, United Kingdom

understandably attracted a great deal of research effort [12, 22]. The aim is often merely to locate the position of a string literal within a larger portion of text, but applications requiring a degree of text-processing ability more sophisticated than this usually offer some form of *pattern* or *template* matching [14], where a pattern is usually defined using strings of characters, some of which have a special meaning (*metacharacters* [20]). The range of such applications is vast, from simple text editors to text-critiquing systems [17], and the means for specifying patterns are equally varied.

The importance of string and pattern matching is reflected by the existence of special-purpose string manipulation programming languages. An example is the work of Griswold *et al.* on SNOBOL4 [16]—although this language has not been without criticism:

> "*One of the most pressing problems in the design of better string manipulation languages arises in the specification of patterns. SNOBOL4 patterns, although extremely flexible and powerful, are notoriously difficult to explain and use.*" [29]

Griswold later echoed these criticisms himself, and suggested the alternative concept of *generators*, used in the *Icon* programming language [15]. More general purpose languages tend not to incorporate pattern matching facilities directly, although they may offer routes to invoking pattern matchers. In C compilers running under UNIX, for instance, string matching is afforded via the *regexp* library routines, which implement patterns specified in the form of *regular expressions*.

Regular expressions are a widely accepted and powerful means of specifying the syntax of string patterns. Their underlying theory is well understood [27], and various implementations of recognizers have been described [20, 24]. The UNIX operating system in particular provides a number of software tools constructed around regular expression recognizers; these include `grep`, `awk` [1] and `lex` [21], and research into extensions to these tools continues (e.g. `agrep` [30] and `TLex` [19]).

Whilst many of the above systems and languages offer a great deal of functionality to the user, the pattern specifications created are often extremely cryptic and terse for all but the most basic of patterns. The differentiation between normal characters and metacharacters is sometimes unclear, and extensive use of *quoting* or *escaping* (e.g. with a backslash character) to convert between the two may serve to add to the confusion. Moreover, pattern symbols are sometimes overloaded, so that they their semantics may vary according to context. It is therefore suggested that the power inherent in these approaches is often masked by the impenetrability of the arcane notational forms.

It has been contended that *visual programming* can make the task of programming more accessible to users by attempting to exploit their non-verbal capabilities [28]. Hence, the possibility arises that a visual implementation of a string pattern notation could be simple to employ, while retaining a high degree of expressive power.

In the CALVIN project [8], a language has been designed and implemented for enabling a tutor (*author*) easily to construct educational applications (*courseware*). The language itself makes use of an entirely graphical notation for specifying program behaviour. However, courseware authoring is one of the many domains for which the analysis of user input plays a key role; often, this input will be textual. Since the string pattern matching problem remains largely unexplored within visual programming, one of the aims of the CALVIN project was to assess the degree to which it could provide a suitable medium for dealing with string analysis.

After presenting an overview of CALVIN, the remainder of this paper elaborates on the pattern matching requirements of the language, describes how these were incorporated, and draws some conclusions regarding the success of the exercise.

2 CALVIN

The acronym CALVIN (Courseware Authoring Language using VIsual Notation) is used to refer not only to the visual programming language, but also to the software development environment that is an integral part of its realization. The language concepts it embodies are few in number, and hopefully easy for an author to use. It can operate in two modes: *student* and *author*. In the student mode, a data structure representing the original authored program is executed by an interpreter. The student's view of this execution is that, as the student follows the interactive lessons, windows are opened, in which diagrams and text appear, answers to questions are solicited through multiple-choice boxes and text-entry boxes etc. The student does not see the visual program code, and is not able to alter it. In author mode, the author can select iconic components; join these together using suitable control-flow constructors such as loops; declare new variables and manipulate and test these in arithmetic and conditional expressions; define and position graphical elements such as windows and menus; and so on.

CALVIN consists of a number of separate but communicating subsystems, the architecture of which is illustrated in Figure 1. Central to these is the Icon Database Management System [6]. This is where icons representing variables and functions are housed and organized, and where it is possible to set up libraries of useful visual code segments.

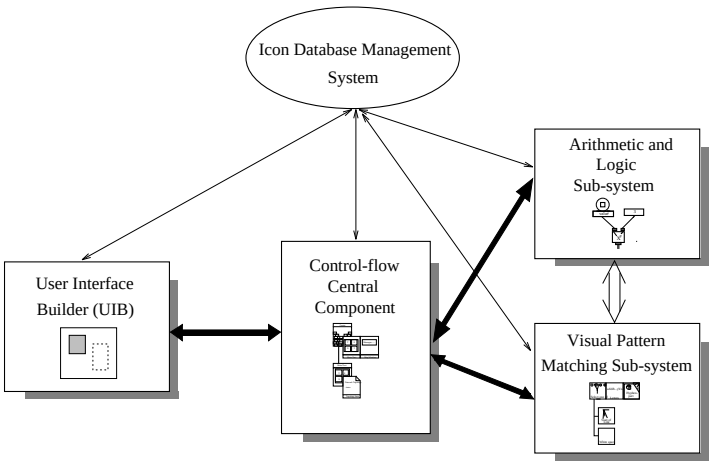

Figure 1: Architecture of the CALVIN System

The Control-Flow component is used to construct the sequence in which material is presented to the student; a typical screen display of this can be seen in Figure 2. A program is built by *dragging* icons onto the programming area. Some icons represent flow of control constructs; others act as operators, and must therefore be associated with iconic operands, which are placed to overlap the operator. Errors of syntax are detected immediately by the system, and the user

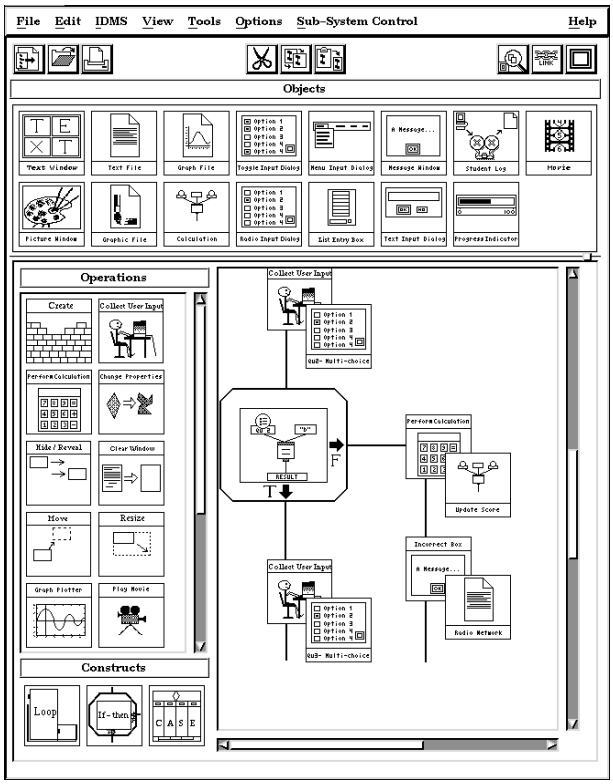

Figure 2: CALVIN Main Control-Flow Window

is alerted to the fact via a message, thus alleviating one of the problems commonly associated with programming in conventional languages. In placing icons on the control-flow window, the author is defining program behaviour at a high level of abstraction: additional information is sometimes required to specify behaviour more fully. Where this is necessary, the User Interface Builder may be invoked. This is a collection of tools, used to detail the properties of graphical objects declared in the program. As a simple example, the author may create a text window on screen by selecting the "Create" operator icon, and pairing this with the "Text Window" operand icon. The User Interface Builder may subsequently be invoked to specify the window's size, position, text font, and so on.

Of the two remaining subsystems shown in Figure 1, the first deals with arithmetic and logic (boolean) expressions, and for this we have adopted a visual data flow approach [13, 18]. Data flow diagrams appear to offer a simple and easily visualised notation for novice programmers [2]. Note that, although it is possible to write complete programs in a data flow fashion, we chose to retain the more usual procedural control flow for the main program. The step-by-step instructional form of most courseware constructed using authoring languages [23] corresponds more naturally to imperative rather than functional formulation, and the multi-paradigm approach it entails is not unusual in visual programming [10].

Finally, there is the pattern matching sub-system. This is invoked whenever the contents of a string variable are to be analysed, particularly when the string represents the input of a user. In determining the sort of pattern matching capabilities this sub-system ought to have, a thorough

survey of facilities was carried out [7].

3 Authoring Language Approaches

One of the best-known authoring languages is PILOT [11]. Its pattern matching mechanism is embodied in the match (M) statement, so that for example:

```
M: UK!BRITAIN!ENGLAND
```

would give a successful match if one of UK, BRITAIN or ENGLAND was entered by the user, and

```
M: BREAD&BUTTER
```

succeeds in the match if both BREAD and BUTTER (in that order) are present in the input. Other options are the ability to specify white space with the '%' character, and a "wildcard" ('*') which will match any single character in the input. In some versions of PILOT a simple form of *fuzzy* matching is also available, specifying that a match may succeed if a single character in a word is mis-typed.

Other authoring languages such as Microtext [4] offer similar facilities, and some languages provide simple extensions. PASS [26], for example, allowed expressions such as:

`toast & -jam` (which matches a string containing toast but not jam)
`+cul+` (which matches any word containing the sub-string "`cul`")

More recent authoring languages and systems have made great advances in the presentational aspects of courseware, but little has been done to improve on the pattern matching facilities of the early languages. Although it was not specifically intended for the purpose, the *information toolkit* HyperCard [5] has received much attention recently. It provides hypermedia capabilities and the means to deliver visually stimulating courseware, but offers little in the way of user input analysis. HyperCard's scripting language HyperTalk, for example, uses verbose but unsophisticated pattern-matching constructs such as:

```
ask "What form is water at 25C?"
if it contains "liquid" then ....
```

To move back to the visual programming arena, the state of the art in authoring languages is probably best represented by commercial products such as Authorware Professional [3]. It must be stressed, however, that Authorware is not a *pure* visual language. While some aspects of courseware—particularly those concerned with the presentation of information—can be programmed using a visual notation, in most applications an author will have to resort to using the supporting *textual* programming language. Thus, for analysing text responses, the author will employ conventional (i.e. non-visual) expressions such as:

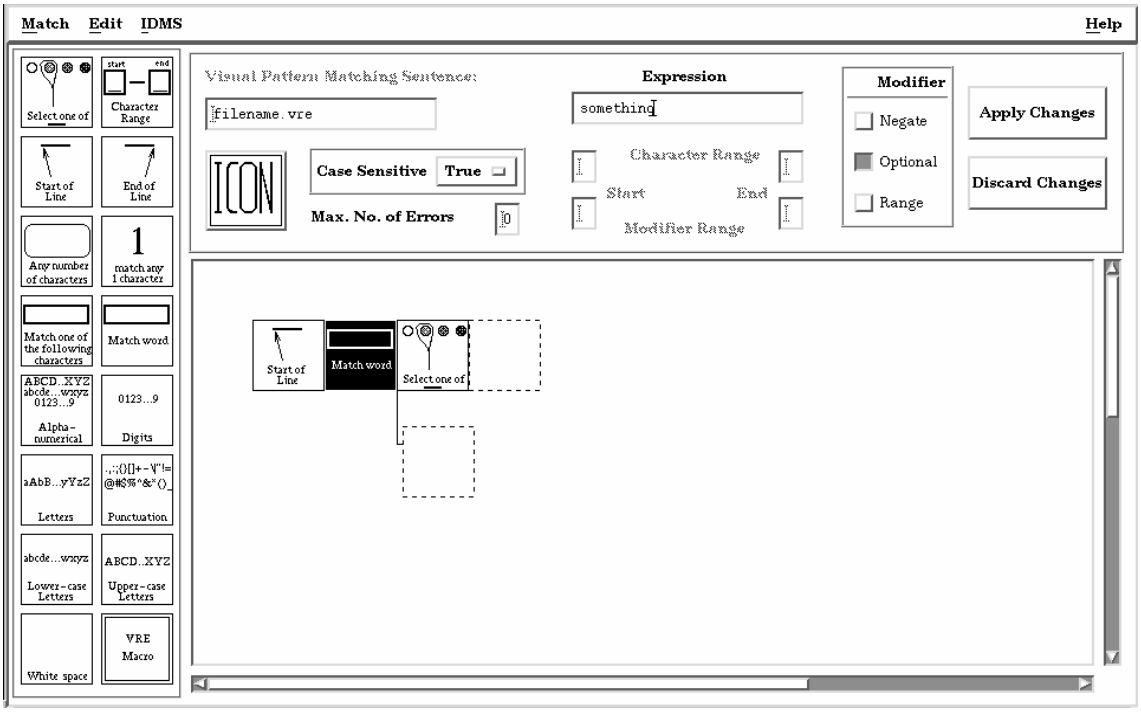

Figure 3: The CALVIN Pattern Matcher

`dog | cat` (which matches "dog" or "cat")
`cat*` (which matches any word starting with "cat")

To summarise, a survey of existing author languages [7] makes it apparent that their pattern matching facilities are largely unsophisticated and lacking in expressive power. Moreover, visual authoring languages offer no visual equivalent of these facilities; this could be construed as either a criticism of visual language designers, or as an indictment of the visual approach itself. Since much of the motivation behind CALVIN is precisely to address this uncertainty regarding the merits of visual programming, the solution adopted has been to take a notation that *does* offer the level of expressive power required—viz. *regular expressions*—and use this as a basis for the CALVIN pattern matcher.

4 Pattern Matching in CALVIN

When an author wishes to specify a text pattern using CALVIN, the pattern-matching sub-system is invoked. The interface, an example of which can be seen Figure 3, is divided into four areas; a menu-bar at the top, a palette of visual pattern matching primitives to the left, a control panel to the right, and a workspace below. Pattern matching expressions are constructed by *dragging* the icon primitives from the palette onto the workspace. A dashed square represents a valid drop position for an icon, although additional mechanisms exist for icon insertion, deletion, copying, and so on.

As mentioned earlier, regular expressions have been used as the foundation for this part of CALVIN, and so most of the available icons correspond directly to primitives adapted from that notation. To be more precise, CALVIN makes use of *extended* regular expressions; these are described in more detail in the following sections, together with examples of their usage. Unless otherwise specified, the term "regular expression" should be taken to mean the extended notation.

4.1 String Literals

The simplest form of regular expression is a string of characters such as "`something`". With CALVIN, the author can specify this using the "match word" icon shown in Figure 4. This is an example of a *parameterised* icon. Parameters are added to a primitive by selecting the required icon, using the mouse button, then editing the appropriate text entry boxes found in the control panel.

Figure 4: The "Match Word" Icon

4.2 Single Unspecified Character

To match any single character, the conventional notation is to make use of a full-stop, so that the pattern "`a.d`" would match "and", "add", etc. The CALVIN equivalent is shown in Figure 5, where the given pattern would match "blender", "blunder", "blinder", and so on.

4.3 Text Positions

In addition to text patterns, regular expressions allow the specification of text *positions*, with '`^`' used to represent the beginning of a line, and '`$`' used to denote line end; thus, "`^hello$`" matches the word "hello" only where it appears on a line by itself. In CALVIN, this expression would be constructed as shown in Figure 6.

Figure 5: Matching an Arbitrary Character

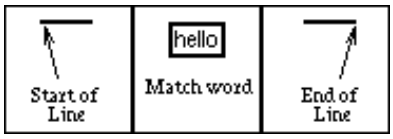

Figure 6: Specifying Positions

4.4 Iterators

A limited form of repetition of expressions is usually provided through the closure operator '`*`' (zero or more instances) and the '`+`' operator (one or more instances). As examples, the pattern "`an*d`" will match "ad", "and", "annd", and so on, while the pattern "`a.+d`" will match any string beginning with 'a', ending in 'd', and containing at least one character in-between. Alternatively, it is possible to specify a lower bound (of at least zero) and an upper bound (of at most 255) on the number of occurrences of an expression. Thus, the pattern "`a.{1,5}d`" is equivalent to "`a.+d`" with an upper limit of five on the number of characters present between 'a' and 'd'. In CALVIN, a single mechanism exists to encompass all of the above: a specific *range* of occurrences (which may involve zero or infinity) of an item can be applied to any portion of an iconic expression. Figure 7(a) shows the representation of our "`an*d`" example, and in Figure 7(b) we see how the "`a.{1,5}d`" example would look.

Figure 7: (a) Representation of "`an*d`"; (b) Equivalent of "`a.{1,5}''`

It is often the case that one wishes to designate a sub-expression as being *optional*; i.e. the match will succeed whether the item appears or not. In CALVIN, this effect could of course be achieved by defining a range with a lower limit of 0, and an upper limit of 1; but where this is to apply to just a single icon, we offer the additional, shorthand facility to specify this by attaching the appropriate icon *modifier*. If the modifier is selected, a question-mark will appear in the top-right corner of the icon, thus acting as a highly visible "optional" marker. For example, in Figure 8 the pattern would match either "telephone" or "phone" on a line.

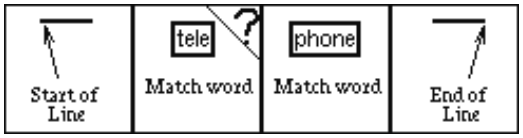

Figure 8: Marking a Sub-Expression as Optional

Figure 9: (a) A Character Class; (b) A Character Range

4.5 Character Classes

A character class conventionally uses square brackets to denote the appearance of any one of the enclosed characters; thus, "`[abc]`" will match the character 'a', 'b' or 'c'. The CALVIN icon for this is shown in Figure 9(a). Where the class covers a consecutive *range* of characters, abbreviations of the form "`[a-z]`" may be used. Another form of parameterised icon is available for such ranges; its use is depicted in Figure 9(b).

An example of the use of the character range class is given in Figure 10. The pattern is used to match octal numbers within a *C* program.

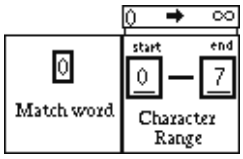

Figure 10: Matching an Octal number in *C*

Conventional regular expressions also allow the *negation* of a character class. Confusingly, the '^' character (also used to denote start-of-line) is normally employed for this, so that "`[^abc]`" matches any character *except* 'a', 'b', 'c' or newline (the latter is negated by default). In CALVIN, another form of *modifier* is used for negation; it appears as a diagonal line across the icon, as shown in Figure 11.

Figure 11: Negating a Character Class

4.6 Alternation

Alternation, i.e. asserting that *one* of several possible sub-expressions will be matched, is normally denoted with the '|' character, so that the expression "`x(a|bc)y`" would match either "xay" or "xbcy". In the CALVIN pattern matcher, the "select one of" icon is used for this purpose, with the various sub-expressions appearing vertically below it, as shown in Figure 12. It is worth noting

that, although this example does not need to use the explicit parentheses present in the text form, a form of bracketing can be achieved in CALVIN by using the mouse to select a group of icons which may then be treated as a single unit. On the user's screen the selected icons are encapsulated within a cross-hatched rectangle, and it then becomes possible to perform operations such as applying an "optional" modifier or a numeric range to the group as a whole. An example of such grouping can be seen in Figure 15 in the concluding section.

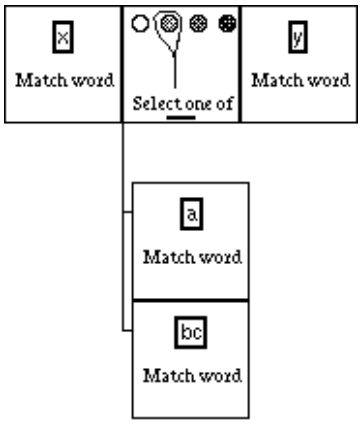

Figure 12: Alternation of Sub-Expressions

4.7 Abstraction

The CALVIN pattern matcher provides many features not normally found in regular expression notations. Some of these, such as the generalised iterator mechanism have already been mentioned. Another powerful and extremely useful feature is the ability to define *meta-icons*. A meta-icon is simply an icon which corresponds to a regular sub-expression, but the programmer or author is free to design the appearance of that icon. Once constructed, a meta-icon may be used in higher-level expressions in the same way as any other icon. Some meta-icons are already built into the system; a few of these are shown in Figure 13.

5 Conclusions

To date, visual programming languages have made their greatest impact in problem domains that are reasonably small and self-contained. The route towards a truly general-purpose visual

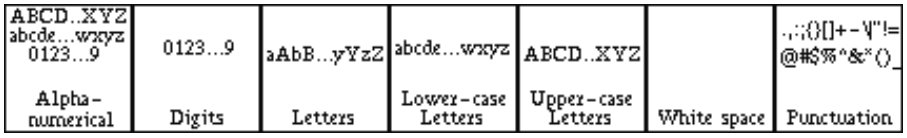

Figure 13: Some of CALVIN's built-in 'meta-icons'

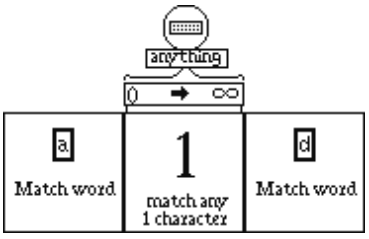

Figure 14: Extracting Matched Text into a Variable

language implies that, sooner or later, language designers are going to have to address the tasks that conventional textual languages are extremely good at solving. Not least among these tasks is that of string processing.

For the CALVIN system, a visual pattern matching notation has been devised which it is felt illustrates that string analysis in a visual programming language is not only feasible, but offers a number of advantages over more conventional notations. For example, it removes the need to remember rules regarding the usage and syntax of metacharacters, there is no overloading of operators, and abstraction may be used to encapsulate any iconic expression as a higher-level icon. A further illustration of the flexibility of the pattern matching system is in its ability to extract the piece of input text corresponding to any portion of a matched expression, and to place this into a variable for further analysis or manipulation. This is shown in Figure 14, in which the (possibly null) string of characters that has been matched between the 'a' and the 'd' is being placed into one of CALVIN's iconic variables.

Further work is undoubtedly required in the design of the icons used within the pattern matching sub-system of CALVIN. The abstract nature of the iconic functions means that constructing a pictorial metaphor to map from the abstract function to the *real world* is difficult [25]. The effort required to design and choose icons for a system is great, c.f. the work of Bewley *et al.* [9] on the Xerox Star. Notwithstanding this, the user testing that has been performed on CALVIN so far leads us to deduce that iconic expressions are much easier to use and to comprehend than the textual equivalents. For example, an extended regular expression to match a numeric literal (e.g. 13, 0, 4598, 1E7 or 3.2e-8) would be:

```
(\+|-)?[0-9]+(\.[0-9]+)?((e|E)(\+|-)?[0-9]+)?
```

whereas the equivalent specification within CALVIN's pattern matcher is shown in Figure 15.

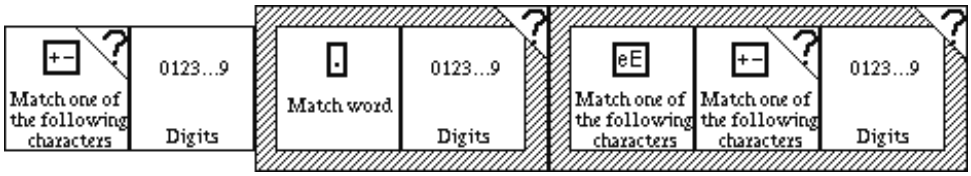

Figure 15: Matching a Numeric Literal

While the visual pattern matcher described here is complete and self-contained, there is still some implementation work to be done on other aspects of the CALVIN language, which may

lead us to make minor modifications and improvements. Once this is finalised, one of our future aims is to use the pattern matcher in the same way as textual regular expressions have been used as the foundation for many UNIX tools, such as `lex` and `awk`. Hence, we hope one day to see a whole host of *visual* programming support tools.

6 Acknowledgements

The authors would like to thank Geoff Kendall, Carolyn Pattinson and Mark Rivers for comments on a previous version of this paper.

References

1. AHO, A. V., KERNIGHAN, B. W., AND WEINBERGER, P. J. Awk — A Pattern Scanning and Processing Language. *Software — Practice and Experience 9* (1979), 267–279.

2. ANJANEYULU, K. S. R., AND ANDERSON, J. R. The Advantages of Data Flow Diagrams for Beginning Programming. In *Proceedings of the International Conference on Intelligent Tutoring Systems: ITS'92* (1992), pp. 585–592.

3. AUTHORWARE INC. *Authorware Professional for Windows: Reference*. 8500, Normandale Lake Blvd., Ninth Floor, Minneapolis, MN 55437, USA, 1991.

4. BARKER, P. G. MICROTEXT: a new dialogue programming language for microcomputers. *Journal of Microcomputer Applications 7* (1984), 167–188.

5. BELL, M. A., AND JACKSON, D. HyperCard: The Perfect CAL Package? In *East-West Conference on Emerging Computer Technologies in Education* (Apr. 1992).

6. BELL, M. A., AND JACKSON, D. Visual Author Languages for CAL. In *Proceedings of the IEEE Workshop on Visual Languages* (Seattle, Washington, Sept. 1992).

7. BELL, M. A., AND JACKSON, D. Authoring Language Mechanisms for Handling Student Responses. In *Proceedings of the International Conference on Computer-Based Learning in Science* (Vienna, Austria, Dec. 1993), pp. 329–338.

8. BELL, M. A., AND JACKSON, D. CALVIN – **C**ourseware **A**uthoring **L**anguage using **VI**sual **N**otation. In *Proceedings of IEEE Symposium on Visual Languages* (Bergen, Norway, Aug. 1993), pp. 225–230.

9. BEWLEY, W. L., ROBERTS, T. L., SCHROIT, D., AND VERPLANK, W. L. Human Factors Testing in the Design of Xerox's 8010 "Star" Office Workstation. In *Proceedings of the CHI'83* (Dec. 1983), pp. 72–77.

10. BORGES, J. A., AND JOHNSON, R. E. Multiparadigm Visual Programming Language. In *Proceedings of the IEEE Workshop on Visual Languages* (Skokie, Illinois, 1990), pp. 233–240.

11. CONLON, T. *PILOT — The Language and How to Use it*. Prentice/Hall International, 1984.

12. DAVIES, G., AND BOWSHER, S. Algorithms for Pattern Matching. *Software — Practice and Experience 16*, 6 (June 1986), 575–601.

13. DAVIS, A. L., AND KELLER, R. M. Data Flow Program Graphs. *IEEE Computer* (Feb. 1982), 26–41.

14. FLECK, A. C. Formal Models for String Patterns. In *Current Trends in Programming Methodology: Volume IV Data Structuring*, R. T. Yeh, Ed. Prentice-Hall, 1978, ch. 8, pp. 216–240.

15. GRISWOLD, R. E., AND HANSON, D. R. An Alternative to the Use of Patterns in String Processing. *ACM Transactions on Programming Languages and Systems 2*, 2 (1980), 153–172.

16. GRISWOLD, R. E., POAGE, J. F., AND POLONSKY, I. P. *The SNOBOL4 Programming Language*, Second ed. Prentice-Hall, Inc, 1971.

17. HEIDORN, G. E., JENSEN, K., MILLER, L. A., BYRD, R. J., AND CHODOROW, M. S. The EPISTLE text-critiquing system. *IBM Systems Journal 21*, 3 (1982), 305–327.

18. HILS, D. D. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages and Computing 3*, 1 (Mar. 1992), 69–101.

19. KEARNS, S. M. TLex. *Software — Practice and Experience 21*, 8 (Aug. 1991), 805–821.

20. KERNIGHAN, B. W., AND PLAUGER, P. J. *Software Tools for Pascal*. Addison-Wesley, 1981.

21. LESK, M. E., AND SCHMIDT, E. Lex — A Lexical Analyzer Generator. Tech. Rep. Computing Science Report 39, Bell Laboartories, Murray Hill, New Jersey 07974, July 1975.

22. PIRKLBAUER, K. A Study of Pattern-Matching Algorithms. *Structured Programming 13*, 2 (1992), 89–98.

23. REYNOLDS, A., AND ANDERSON, R. H. *Selecting and Developing Media for Instruction*, 3rd ed. Van Nostrand Reinhold, 1992.

24. RICHARDS, M. A Compact Function for Regular Expression Pattern Matching. *Software — Practice and Experience 9* (1979), 527–534.

25. ROGERS, Y. Evaluating the meaningfulness of icon sets to represent command operations. In *Proceedings of HCI'86 Conference on People and Computers* (1986), British Computer Society, pp. 586–603.

26. ROTHWELL, J., Ed. *Authoring Systems*. CBT Library, Module 2. National Computing Centre Limited, 1983.

27. SEDGEWICK, R. *Algorithms*, 2nd ed. Addison-Wesley, 1988. (Pages 293-303).

28. SHU, N. C. *Visual Programming*. Van Nostrand Reinhold Co., 1988.

29. STEWART, G. F. An Algebraic Model for String Patterns. In *Proceedings of the Second Symposium on Principles of Programming Languages* (Palo Alto, CA, 1975), pp. 167–184.

30. WU, S., AND MANBER, U. `agrep` — A Fast Approximate Pattern-Matching Tool. In *Proceedings of Winter USENIX'92 Conference* (1992).