Introduction

One point of view of programming languages takes a programming language to be a set of primitive elements, one or more means of combination of the primitive elements, one or more means of abstraction, and a control structure for unwinding the abstractions of the language giving the abstractions effective, procedural interpretations. From this point of view, a programming style reflects the behavior of the control structure of the interpreter of a language, and defining a new style involves defining an entirely new control structure or modifying or augmenting the control structure of an existing interpreter as well as providing language-level facilities for either explicitly or implicitly accessing the new features of the extended control structure. For example, adding a mechanism for making choices at programmer-defined branch points, searching, and backtracking from failed choices defines the nondeterministic style of programming, and incorporating into the control structure of a language effective, procedural interpretations of the operations of propositional calculus, and, or, and not, defines logic programming as a style. Given all of the possibilities for manipulating and processing data, one can take a purely abstract approach to defining new programming styles by considering the possibilities for primitive elements and control not yet incorporated into languages as control mechanisms or made visible to programmers at the level of language and design control and language extensions involving previously unused or incompletely explored possibilities.

A Pattern-Matching Control Extension

As a mechanism for the control of evaluation of the expressions of a language, pattern-matching has a long history. Interpreters for dialects of LISP and Scheme often use pattern-matching to compare the list of formal parameters of a lambda expression with the list of actual arguments and generate an exception at runtime if the formal parameters and actual arguments do not at least agree in number. The interpreters of logic programming languages like Prolog use pattern-matching in their implementations to generate and filter streams of assertions in the course of solving a problem. While pattern-matching has a long history in the interpretation of languages, the common uses of pattern-matching in interpretation give the programmer only implicit access to pattern-matching and employ pattern-matching only as a filter or guard on data rather than generatively. One could aim to use the characteristic of pattern-matching that patterns do not match data uniquely to cause branching in computation, and I proposed to extend a simple, functional language with control mechanisms using pattern-matching and to make language-level changes to the language to give a programmer explicit access to the pattern-matching control mechanism.

Strategy for Implementation

I chose to use the Scheme language to implement my pattern-matching extension to the control structure of a simple, functional language. I chose Scheme because with its simple, regular syntax, the evaluation of Scheme is easily described meta-circularly making the construction of an interpreter for the proposed language extension straightforward. After implementing a basic, meta-circular evaluator and augmenting the meta-circular evaluator with crude matching functions that take a Scheme symbol and compare the symbol
with an argument string testing for containment of the symbol in the string, I intended to add a keyword, `match`, taking a symbol as argument and using the internal matching functions to scan the environment of invocation for names matching the argument symbol. Recognizing that Scheme only accepted a procedure in the operator position of an S-expression, I believed that the opportunity existed to coherently extend the evaluation semantics of Scheme to permit other types of entities in the operator position of an S-expression, and I intended to make pattern-matching convenient and useful by permitting the use of match results as operators.

The Blather Interpreter

The Blather interpreter represented the culmination of my efforts to integrate pattern-matching into a simple, functional language. To support the extension of the evaluation semantics of Scheme, I added a new, tagged-type to the set of data types supported by the meta-circular evaluator, which consisted exclusively of primitive and compound procedures relying on the underlying implementation to tag and handle all other primitive elements of the language. The `match` keyword invoked a function in the evaluator to match its argument against names in the environment of invocation matching all unique names in the environment in the absence of an argument pattern and return a match object, a tagged list of names and the environment in which the match occurred. The tag on the match object allowed me to extend the `apply` function of the meta-circular evaluator to treat match objects as operators. Finally, to allow the programmer to recover the results of matches either as bare lists of names or values for the purpose of explicitly processing match results, I added two special forms, `match-object-to-names`, which takes a match object as an argument and extracts the list of names formed by the match operation, and `match-object-to-values`, which takes a match object as argument and evaluates all of the names in the match object against the environment of the match object. In summary, we added the following keywords and special forms:

- `(match [<symbol>])` returns a match object consisting of a tagged list of all unique names in the environment of invocation matching `symbol` or all names in the environment if called without an argument;
- `(match-object-to-names <match_object>)` returns a bare list of names as symbols from the argument `match_object`;
- `(match-object-to-values <match_object>)` returns a list of the results of evaluating each name in `match_object` in the associated environment of `match_object`;
- `(match_object . <argument_list>)` resolves each name in `match_object` against the associated environment of `match_object` applying each result in turn to `argument_list` and concatenating the results of each application.

Features of the Interpreter

As designed and implemented, explicit matching of patterns against names in the environment proved to be orthogonal to the existing features of Scheme and easily composable with other features of the language. For programs that did not use any of the language extensions, the augmented interpreter behaved as a normal Scheme interpreter; Blather worked as a backward-compatible extension to Scheme. The matching operation and the resulting match objects enjoyed full, first-class status in the language; matching could be used in any place in which one could use `lambda`. Though the extension worked generally, as a practical matter I noted that programmers might find the match objects to be difficult to use as operators—the name of a function against which a match would be made guarantees nothing about the structure of the expected arguments of the function—but when coding in a uniform style permitted by the Scheme interpreter but eschewed by most programmers in which each lambda expression takes only a single argument representing a list and assumes responsibility in the body of the lambda expression for parsing and interpreting the argument list, programming with match objects has the potential to become robust and useful.
Possible Applications

As I developed my extension to Scheme as a purely abstract exercise in generating a new style of programming by augmenting the control structure and language-level features of a language, I had no specific applications in mind for the resulting extension. However, I believe that the pattern-matching style of programming made possible by my extension could find applications in any problem domain in which the entities in a typical problem couple names closely with data or behaviors and in which small changes to the forms of names translate to small changes in data or behavior. For example, in the analysis of DNA and RNA and protein-folding, the base pairs in a sequence of DNA or RNA correspond to operations in the construction of a protein with some sequences of base pairs naming amino acids to be used in the construction of a protein and other sequences of base pairs indicating transformations of the protein molecule under construction. The sequences of base pairs in a segment of DNA or RNA name a set of data and operations and as names correspond to code-carrying labels.

In conventional programming languages, either a program works toward an explicit goal, or a program contains explicit stopping conditions, conditions that implicitly describe the goal of computation. By reconceptualizing programming to involve using the host style, the functional style in the case of the Blather interpreter, of a pattern-matching interpreter to describe a strategy for program evolution by the generation of names and the mutation of the code associated with names by reflection and to express stopping conditions for a program implicitly rather than explicitly, we might be able to develop a new, pattern-directed style of programming. Since patterns represent a bridge between symbolic and subsymbolic knowledge, a pattern-directed style of programming might offer new approaches to the open problem of extracting symbolic knowledge from correctly functioning subsymbolic processing systems; a coupling of code to names and the association of names through patterns might permit the examination of the state of a correctly functioning subsymbolic system for the purpose of extracting an algorithm for the behavior of the system by analysis of the relationships between names in the system and examination of the code associated with each name.

Future Work

As I had only a relatively short time in which to conceive and implement my idea, much work remains to be done. First and foremost, I would like to implement a large system using my pattern-directed ideas through the Blather interpreter with the goal of studying the expressiveness and usability of the pattern-directed approach to programming. In the course of implementing a large system with the Blather interpreter, I would naturally need to consider the fitness for the purpose of pattern-directed programming of the internal data structures, the keywords, and the special forms of the augmented interpreter and language. While implementing a large system, shortcomings of the existing extensions to the language would reveal themselves suggesting new features to support the augmented interpreter and the associated style. Of course, the idea of pattern-directed programming may prove too uncontrollable or too inexpressive for the development of an actual system, but only by making a serious effort to develop a large system in a pattern-directed style could I hope to evaluate the potential of pattern-directed programming as a style.

Reflections on the Project

I made orderly, methodical progress in the development of the Blather interpreter, and though an abstract approach analyzing and extending language control structures might seem of questionable value, I think that I have vindicated the approach by implementing an original language extension and, more importantly, suggesting new directions in programming as a result of my work on the extension. Though I would have liked to make real progress on a large project and test my ideas for new programming styles, the short duration of the course set low limits on the possibilities for my exploration. I have had to remain content to present an implementation of an interpreter for my control and language extension while my style ideas remain abstract, and I ask for the forebearance of the reader.
Conclusion

While there may exist other possible views of programming languages and styles, the view of a language as defined by the interpreter of the abstractions of the language and the view of a programming style as proceeding from the features of an interpreter of the style offer a firm foundation for the critical study and extension of existing programming languages and styles as well as for the invention of entirely new styles of programming, and by assuming this view of languages, I identified pattern-matching as an incompletely developed control mechanism only finding implicit use in programming languages. In looking for ways to make pattern-matching explicitly available to the programmer, I had the idea to use pattern-matching reflectively by allowing a programmer to invoke matching against the environment of a process, the names used by the running program. While I initially considered the extension to be just a novelty, after implementation and some crude experimentation, I speculated that pattern-matching as an explicit act in programming might be the basis for a new style of pattern-directed programming in which programs evolve reflectively creating names, patterns, and functions in the pursuit of programmer-defined, implicit terminating conditions. Though I had insufficient time to explore the possibilities for solving problems in a pattern-directed style and to evaluate the potention of such a style, I trust that the reader will regard the generation of new ideas as a valuable end in itself, and I look forward to continuing my investigation into pattern-matching as a way to organize and capture knowledge, the essence of any style of programming.