An Interrogative Approach to Novice Programming

Alexander Quinn
Department of Computer Science and Engineering
University of Washington
aquinn@cs.washington.edu

Abstract

Domain specific programming languages tend to be rigid in capability and dependent on either a graphical interface or a scripting language. We present a question-oriented approach that requires no prior knowledge of programming and can be easily adapted to a wide range of domains. Interrogative programming works by "parsing" the user's intent using the responses to a series of closed-ended questions. Questions are guided by a context free grammar specified in an external file. We discuss the benefits, capabilities and limitations of interrogative programming along with the results of recent usability studies with our prototype.

1. Introduction and Related Approaches

The goal of end user programming is to empower users who have no prior programming experience to solve simple problems with computers or customize existing applications to their own ends, without teaching them formal programming [1].

For users, the difficulty is in bridging their concept of a solution with the computer's accepted means of instruction. Typically, this problem is solved with either scripting languages, programming by example [3], or visual programming.

Scripting languages, such as GUILE, AppleScript, or Matlab try to make the language more accessible by abstracting away details such as memory management and complicated data structures. Users who invest the time to learn are rewarded with increased power and flexibility in their ability to customize applications or even create new ones. However, this still requires that the user learn a syntax and basic programming strategies. Thus, scripting languages are usually inappropriate for true novices.

Programming by example (PBE), another common approach, allows the user to perform actions representative of a solution and have the computer generalize those actions to create the desired automation. PBE bridges the gap between concept and specification by allowing the user to naturally select the data to be operated on in a graphical environment [4] and communicate the desired operations in an environment the user is already familiar with. However, it usually limits the kinds of programs that can be created to repetition of tedious tasks.

Visual languages are a class of programming languages that allow the user to use diagrams or other visual means to specify the solution to a problem. While they often provide a more intuitive and richer means of specification than conventional textual languages, developing the right visual abstractions can be difficult so they can be hard to scale to large applications [2].

We discuss how some of the limitations of each of these can be overcome through a new approach called *interrogative programming*. Interrogative programming is a means of *extracting* the user's intent through a line of questioning based on closed-ended questions. We show how this method offers generality, accessibility, and flexibility to end-user programming.

Throughout this paper, we use the terms *end-user programming* and *novice programming* interchangeably. Novice programming is programming in an environment that makes programming more natural for non-programmers [1]. End-user programming is programming by either programmers or non-programmers where the programming is not the focus of the activity. Interrogative programming can fit into both categories.

2. Description of Interrogative Programming

Interrogative programming tries to discover what the user wants by asking a series of closed ended questions. An answer will be either a selection from a list of choices or the raw input of a string or number. Each choice clarifies some aspect of the program. While the approach is not domain specific, a particular implementation is.

2.1. Example: Print the Numbers 1 to 10

The following example shows how a toy implementation could be used to create a program to count to 10. Useful programs are certainly possible, but small examples like this are best for illustration. This is a condensed version of what you would see in a real interrogative programming system.

Computer: This system lets you create trivial examples.
Does this sound like what you want?
1. Yes
2. No

User: 1
Computer: To create a trivial example, you need to specify one or more instructions. Please pick one.
1. Print something to the screen.
2. Set or change the value of a variable.
3. Repeat some actions a certain number of times.
User: 3
Computer: Enter the number of times to repeat.
User: 10
Computer: You need to specify the first instruction for the program to repeat. Please pick one.
1. Print something to the screen.
2. Set or change the value of a variable.
3. Repeat some actions a certain number of times.
User: 1
Computer: To print something to the screen, you need to specify what to print. Please pick one.
1. A particular string of text.
2. The value of one of your variables.
3. Today's date.
4. The number of times the actions have been repeated.
User: 4
Computer: You may specify another instruction for your loop if you like. Please pick one.
1. Print something to the screen.
2. Set or change the value of a variable.
3. Repeat some actions a certain number of times.
4. Stop specifying actions.
User: 4
Computer: You may specify another instruction for the program if you like. Please pick one.
1. Print something to the screen.
2. Set or change the value of a variable.
3. Repeat some actions a certain number of times.
4. Stop specifying actions.
User: 4
Computer: Your program is almost complete. Please enter a file name for your program so you can run it later.
User: count
Computer: Thank you. Your program is complete.

Notice that the system would allow the user to create control flow that would be difficult or impossible to achieve with a PBE system. Also, all of the questions are in English and the user always knows what the system is capable of. This is an unappreciated feature of dialog based systems.

2.2. Theoretical Foundations

How is this different from an application wizard? With interrogative programming, all of the questions are based on a context free grammar that describes the kinds of programs that can be created with a given implementation. For example, in the previous example, the third question is based on a production like this.

```
STATEMENT ::=  PRINT_STMT |
               ASSIGNMENT_STMT |
               LOOP
```

The process of creating a program is similar to the way a top down recursive-descent parser analyzes the syntax of a traditional program except that instead of deciding productions based on a lexeme stream, it asks the user questions to determine which non-terminals to take.

3. IPS: A Prototype

We have constructed a console-based prototype called IPS (Interrogative Programming System) that allows a user to create a stand-alone executable by answering a series of closed-ended questions, similar to the example above.

Besides the user's input, IPS takes a special "domain file" as input. A domain file is a specification of a context free grammar along with other information to guide the dialog and code generation. The use of domain files allows the same system to be easily adapted to many different domains. Domain files work much like the grammar files that are used with yacc-style parser generators.

A domain file could be written by any programmer to suit some application. Just as a yacc user need not know the inner details of yacc to create a parser, it is not necessary to understand the workings of IPS to write a domain file. The domain file specifies the grammar, the skeleton code, and English phrases to support the dialog. IPS does the interaction and intent parsing. Domain files must be

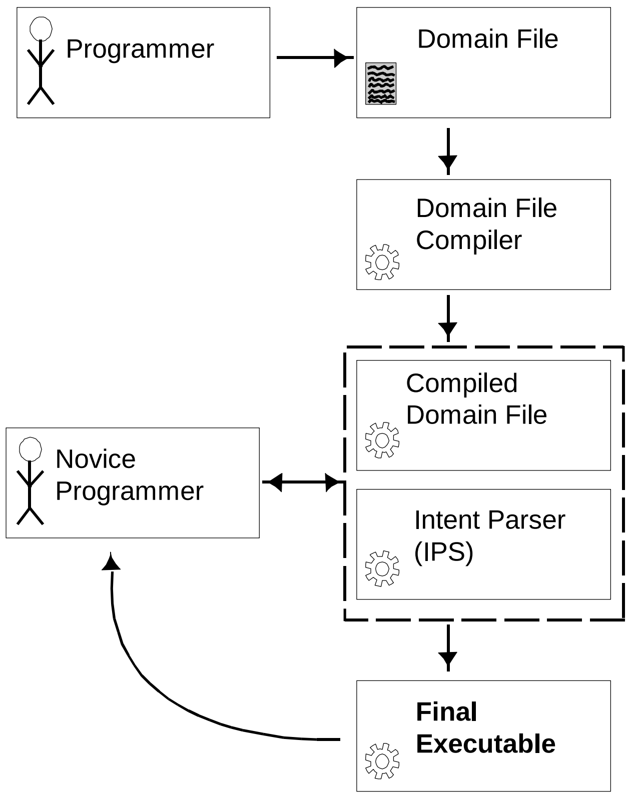

Figure 1: IPS architecture

compiled into a special “compiled” form and checked before they can be used. Figure 1 shows how this all fits together.

IPS generates a file with source code. Like a compiler, it first builds an abstract syntax tree (AST) as it *parses* the user's intent. At the end, the AST is evaluated and a file is written with a name specified by the user. Much like in a yacc grammar, the domain file contains code with placeholders like "$1" and "$2" which are replaced by synthesized attributes.

It is important to note that IPS is independent of the target programming language. For example, we have created domain files that output Python, Java, C, and HTML files.

4. Usability Studies

Two formal usability studies were conducted to evaluate the usability and expected utility of our prototype. A total of 13 subjects were paid $10 each to meet a researcher individually at an academic research lab and voluntarily spend an hour working with IPS. All subjects were adults, ages 18 to 55 who used a computer at least 5 days per week but had never written a computer program in any language that supports loops or other control flow. The two usability studies were done with very similar methodology and conditions. Usability improvements were made to IPS after the first study. We discuss only the results of the second study which used 10 of the participants.

In the second study, subjects were given three simple programming tasks to accomplish using IPS. They were asked to create a “Hello world” program, create a program to print the average of two integers supplied by the user of the program, and create a program to print a pattern with asterisks.

We were specifically trying to test whether or not people could engage in a problem with such a system. Performance was fairly low, but we did see some promising results. All subjects solved the first problem without any difficulty. Only one subject (10%) solved the second problem, computing the average of two numbers. This low performance was expected because the wording and structure of the questions were very close to real programming, with references to integer variables and such. The most common problem we saw was that subjects were focused on the final objective of printing an average, rather than the steps necessary to collect the information and then process it. Four (40%) subjects chose as the first action in the program, an option to "Do a calculation" where they should have chosen an option to "Collect some keyboard input from the user." The third problem, printing the design, did not use the programming terminology. Five subjects (50%) solved it without any assistance and four (40%) solved it with a small hint.

People seemed to understand the basic idea and all subjects were able to make substantial progress toward at least two of the problems. One reason for the low performance might be that they felt pressured because the test was only an hour long. One subject noted that if he were using such a system at home rather than in a test setting, he would be more likely to explore help features.

5. Conclusion and Future Work

We have made substantial progress in developing interrogative programming as a paradigm and demonstrating it with a prototype. However, we have several unsolved problems which will be explored in future research.

First, there is currently no way to edit programs. Without this, a user can only write programs which can be made correctly the first time. We are considering both graphical solutions and natural language solutions to this.

Also, in order to be truly useful, interrogative programming would have to support functional decomposition. Allowing the user to create functions would require different kinds of dialog because the dialog would have to switch between the context of the “main” function and a function which was being defined.

Finally, the current system forces the novice programmer to solve all problems in a depth-first manner. Until one step is fully specified, the novice programmer cannot go on to the next step. We have looked at graphical solutions to this as well as simply allowing the user to defer answering a question.

6. Acknowledgments

We gratefully acknowledge Steven Tanimoto and Jeremy Moody for help in developing the concept and this paper, Batya Friedman for providing advice and space for the usability studies, and the thirteen anonymous usability study participants for their time and constructive feedback.

7. References

[1] D. Gilligan, "An Exploration of Programming by Demonstration in the Domain of Novice Programming", Master's thesis at Victoria University of Wellington, Wellington, New Zealand, August 1998.

[2] R. Jamal, L. Wenzel, “The Applicability of the Visual Programming Language LabVIEW to Large Real-World Applications”, Proceedings of the 11th International IEEE Symposium on Visual Languages, Darmstadt, Germany, p. 1, 1995.

[3] D. C. Smith, A. Cypher, L. Tesler, "Novice Programming Comes of Age", *Your Wish Is My Command: Programming by Example*, Morgan Kaufmann, San Francisco, February 2001.

[4] R. St. Amant, H. Lieberman, R. Potter, L. Zettlemoyer, "Visual Generalization in Programming by Example", *Your Wish Is My Command: Programming by Example,* Morgan Kaufmann, San Francisco, February 2001.