Roles of Variables and Learning to Program

Jorma Sajaniemi
University of Joensuu, Department of Computer Science
saja@cs.joensuu.fi

ABSTRACT
Computer programming is a difficult skill for many students and new methods and techniques to help novices to learn programming are needed. Roles of variables is a recently introduced concept that captures expert programmers' tacit knowledge in a way that can be explicitly taught to students. The use of roles and role-based program animation in teaching has been found to facilitate learning programming skills. During this talk I will introduce the role concept and the individual roles, describe how roles can be introduced to students, and review empirical results obtained by our research group while investigating the applicability and effectiveness of roles.

KEY WORDS: Computer programming, Roles of variables, Programming skills

INTRODUCTION
To learn computer programming is difficult for many students. Programs deal with abstract entities—formal looping constructs, pointers going through arrays etc.—that have little in common with everyday issues and are therefore hard to learn. These entities concern both the programming language itself and the way programming language constructs are assembled to produce meaningful combinations of actions in individual programs. The concept of variable is very difficult for students. For example Kolikant & Haberman (2001) found that given the statements:

```
read(A, B);
read(B);
write(A, B, B);
```

many students were not at all sure what happens when reading twice to the same variable or writing twice from the variable. Other examples of problems with variables are reported by, e.g., Ben-Ari (2001), Holland et al. (1997), Samurçay (1989), and Sleeman et al. (1989).

So far, efforts to ease and enhance learning have varied in their general approach to improve learning: most studies report effects of new teaching methods and new ways of presenting teaching materials, while the introduction of new concepts has been far more rare. There are only few examples of research into new concepts that can be utilized in teaching introductory programming: software design patterns, and roles of variables. Software design patterns (Clancy & Linn 1999) represent language and application

Proceedings of the 3rd Panhellenic Conference «Didactics of Informatics»
A. Jimoyiannis (ed.)
University of Peloponnese
Korinthos, Greece, 7-9 Oct. 2005

independent solutions to commonly occurring design problems. The number of patterns is potentially unlimited, and there are sets of patterns for various levels of programming expertise (e.g., elementary patterns for novice programmers (Wallingford 2003)) and application areas (e.g., data structures (Nguyen 1998)). Research into the use of patterns indicates that there is a need to regularly refine patterns used in teaching (Clancy and Linn 1999).

Roles of variables (Sajaniemi 2002, 2005) describe stereotypic usages of variables that occur in programs over and over again. Only ten roles are needed to cover 99 % of all variables in novice-level procedural programming (Sajaniemi 2002), and they can be described in a compact and easily understandable way. As opposed to the patterns approach, the set of roles is so small that it can be covered in full during an introductory programming course. Furthermore, program animation can be based on roles resulting in animation of programming concepts—how variables are utilized to create meaningful functionality—as opposed to the animation of programming language constructs. According to Petre & Blackwell (1999), visualizations should not work in the programming language level because within-paradigm visualizations, i.e., those dealing with programming language constructs, are uninformative. The role concept provides an opportunity to make informative visualizations for learning programming.

This paper describes how roles can be introduced to novices and why it should be done. The rest of the paper is organized as follows. Section 2 provides motivation by reviewing a classroom experiment where roles were introduced during an elementary programming course. The role concept is then described in Section 3; followed by instructions for utilizing roles in teaching in Section 4. Section 5 reviews some of our research on the role concept: role knowledge in expert programmers' mental models, CS educators' opinions of roles, and roles in different programming paradigms. Finally, Section 6 contains the conclusion.

CLASSROOM EXPERIMENT

We have conducted a classroom experiment during an introductory Pascal programming course. The focus of the experiment was on the use of roles and role-based animation in teaching (Sajaniemi & Kuittinen 2005, Byckling & Sajaniemi 2005). The participants of the experiment—ninety-one Finnish undergraduate students studying computer science for the first semester—were divided into three groups that were instructed differently: in the traditional way in which the course had been given several times before, i.e., with no specific treatment of roles (*the traditional group*); using roles throughout the course (*the roles group*); and using roles together with the use of a role-based program animator, PlanAni (*the animation group*). The course lasted five weeks, with four hours of lectures and two hours of exercises each week. During exercises, all groups animated four programs. For animation, the animation group used PlanAni and the other groups used a visual debugger (Turbo Pascal v. 7.0) with all variables added to the watch panel of the debugger. The lectures were based on existing materials not specially designed for the introduction of roles; this decision was based on our intention not to interfere with the teaching of the traditional group and to present all groups as similar teaching as possible.

Participants attended an examination after the course. Their answers were graded for the purposes of the course but, in addition, they were further analyzed for the purposes of the experiment. One of the tasks in the examination was to write a summary of a given short program. A preliminary analysis of the grades given by the teachers revealed that teachers' grading did not reflect students' understanding well. To analyze this finding further, we selected all answers having no errors and demonstrating full understanding of every detail of the program, and calculated group grade means for them. As the maximum grade was 6.0 one would have expected that all the selected answers would have the grade 6.0. However, for the traditional group the grade mean was 4.6, for the roles group 5.0, and for the animation group 4.1. As all answers selected into the new analysis demonstrated complete understanding of the program, the differences in grade means imply differences in the way students described the program.

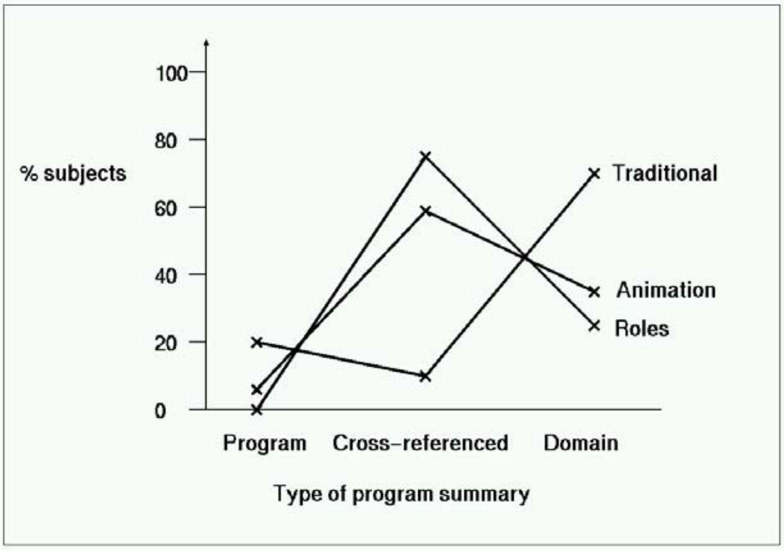


Figure 1: Distribution of subjects with different program summary types determined by the amount of domain and program information in object descriptions (Sajaniemi & Kuittinen 2005)

All program summaries were then analyzed using Good's program summary analysis scheme (Good & Brna 2004) that looks at the level of expressions used, e.g., whether objects were referenced in program terms ("variable w") or in domain terms ("patient's weight"). There were major differences in the distributions of expression levels between groups: program summaries in the traditional group had either a small or a large number of domain statements while in the other two groups program and domain statements were used more evenly. To analyze this difference further, we used a similar method as Pennington (1987) and sorted program summaries into three types—*program-level summaries*, *domain-level summaries*, and *cross-referenced summaries*—depending on the amount of program vs. domain statements. Figure 1 shows the distribution of summary types among each group. The number of cross-referenced summaries was significantly smaller among the traditional group than among the other groups (x^2=10.773, *df*=2, *p*=0.0046). In Pennington's study, high comprehension programmers almost uniformly used cross-referenced summaries while low comprehension programmers tended to produce either a program-level summary or a domain-level summary. Our result thus indicates that roles provided students a conceptual framework that enabled them to mentally process program information in a way similar to that of good code comprehenders. However, the teachers in our experiment gave better grades for poorer understanding. One may wonder, whether this behavior is common in programming teachers who usually are ignorant of even the most central results of psychology of programming.

At the end of the course we videotaped some student pairs when they constructed a new program. In this task the animation group outperformed the other two groups: the percentage of correctly implemented subtasks was 92 % for the animation group as compared to 44 % for both the traditional group and the roles group. Moreover, all pairs in the animation group used an optimal set of variables needed for the task whereas no pair in the other groups used the optimal set.

We analyzed also the fluency of programming activities by examining the writing order of program code lines observed in the videos. The basis of this analysis was Rist's model of schema creation in programming (Rist 1989, 1991). According to Rist, programming process consists of implementation of *program plans*, which either have to be retrieved from memory, or created during programming. Rist has devised a model that can be used to analyze whether a programming process indicates the possession of plan knowledge, which results in *forward development* of the program, or lack of this knowledge, which results in less efficient *backward development*. We extended Rist's model to cover variable plans in detail and analyzed the videotaped programming protocols. The result of the analysis is given in Figure 2 where each student pair is depicted by a small circle. The animation group exhibits the largest amount of forward development, i.e., they have the required knowledge and they can apply it fluently. The other two groups exhibit mainly backward development that results in poorer performance.

The analysis of the program *comprehension* results (Figure 1) showed that the mental models of both the roles group and the animation group were better than those of the traditional group. Thus the increased programming knowledge—given in the form of roles—enhanced the construction of program knowledge in a program comprehension task. On the other hand the analysis of program *creation* results (Figure 2) showed that only the animation group performed well. This suggests that the animator elaborated the increased programming knowledge so that the students could use it successfully also in program construction. This, along with the best performance of the animation group suggests that role knowledge should be elaborated by role-based program animation in introductory-level teaching.

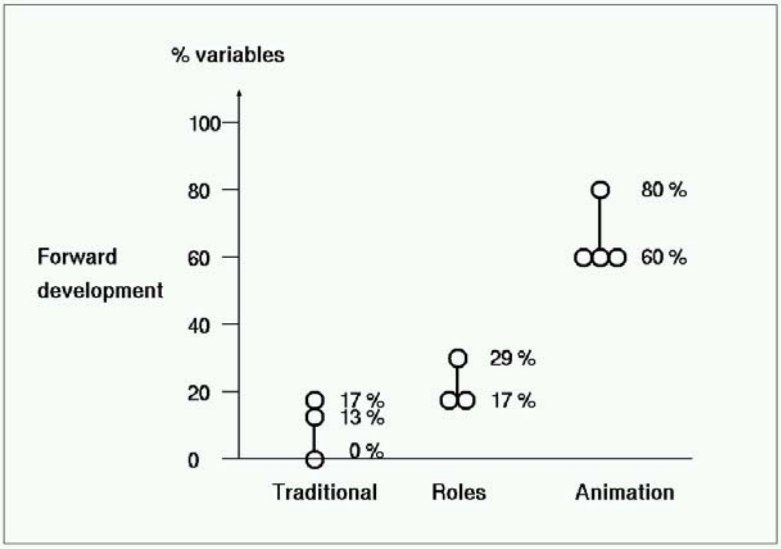


Figure 2: Percentages of forward development by each student pair in program construction (Byckling & Sajaniemi 2005)

ROLES OF VARIABLES

The notion of the *role of a variable* is based on the fact that variables are not used in a random or *ad-hoc* way but there are several standard use patterns that occur over and over again (Ehrlich & Soloway 1984, Rist 1989, Green & Cornah 1985). In programming textbooks, two patterns are typically described: the counter and the temporary. I have developed this idea further (Sajaniemi 2002) and characterized the role as the behavior of a variable, i.e., the sequence of its successive values. In this definition, the way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment.

For example, consider the Pascal program in Figure 3, which contains three variables: data, count, and value. In the first loop, the user is requested to enter the number of values to be later processed in the second loop. The number is requested repeatedly until the user gives a positive value, and the variable data is used to store the last input read.

The variable value is used similarly in the second loop: it stores the last input. There is no possibility for the programmer to guess what value the user will enter next. Since these variables always hold the latest in a sequence of values, we will say that their role is that of *most-recent holders*. The variable count, however, behaves very differently. Unlike the other variables for which there is no known relation between the successive values, once the variable count has been initialized, its future values will be known exactly. The role of this variable is that of a *stepper*.

```
program doubles;
   var data, count, value: integer;
begin
   repeat
      write('Enter count: '); readln(data)
   until data > 0;
   count := data;
   while count > 0 do begin
      write('Enter value: '); readln(value);
      writeln('Two times ', value, ' is ', 2*value);
      count := count - 1
   end
end.
```

Figure 3: A simple Pascal program

Table 1 lists ten roles that cover 99 % of variables in novice-level procedural programs (Sajaniemi 2002) and gives for each role an informal definition suitable to be used in teaching; exact definitions of the roles can be found in the Roles of Variables Home Page (Sajaniemi 2005). In this role set, an array is considered to have some role if all its elements have that role. For example, an array is a *gatherer* if it contains 12 *gatherers* to calculate the total sales of each month from daily sales given as input. Moreover, there is a special role for arrays—*organizer*.

The role of a variable may change during the execution of a program and this happens usually somewhere between two loops. For example, in the program of Figure 3, the two variables data and count could be combined to a single variable, say count (making the assignment “count:=data;” unnecessary). The role of this variable would first be a *most-recent holder* and then, in the second loop, a *stepper*.

Roles are cognitive—rather than technical—concepts. For example, consider the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ... where each number is the sum of the previous two numbers. A mathematician who knows the sequence well can probably see the sequence as clearly as anybody sees the sequence 1, 2, 3, 4, 5, ..., i.e., the continuum of natural numbers. On the other hand, for a novice who has never heard of the Fibonacci sequence before and who has just learned how to compute it, each new number in this sequence is a surprise. Hence, the mathematician may consider the variable as stepping through a known succession of values (i.e., a *stepper*) while the novice considers it as a *gatherer* accumulating previous values to obtain the next one.

Table 1: Roles of variables in novice-level procedural programming

Role	**Informal description**
Fixed value	A variable initialized without any calculation and not changed thereafter
Stepper	A variable stepping through a systematic, predictable succession of values
Follower	A variable that gets its new value always from the old value of some other variable
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far
Gatherer	A variable accumulating the effect of individual values
Transformation	A variable that always gets its new value with the same calculation from values of other variables
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed
Temporary	A variable holding some value for a very short time only
Organizer	An array used for rearranging its elements

In addition to teaching programming, roles have other applications. For example, automatic analysis may be used to find roles in large programs and thus assist maintenance engineers in program comprehension.

ROLES IN TEACHING PROGRAMMING

Roles are not just a collection of additional concepts that enlarges the amount of material to be learned but—as found in Section 2—they are an instrument for thinking. Roles help students not only to understand the life cycle of a variable but both to design and to mentally process programs in a better way. Roles should not be taught as a separate issue but introduced gradually one by one as they appear in example programs of a programming course. This section describes how roles can be taught and how they can be utilized in teaching programming strategies. It is based on our earlier article (Kuittinen & Sajaniemi 2004).

Role Knowledge Construction

The constructivistic approach to learning, e.g. Bereiter (1994), von Glasersfeld (1995), stresses that new knowledge can be effectively acquired only if it is actively built on the top of existing knowledge. Figure 4 depicts connections between roles and some programming language constructs and provides a basis for constructive learning. Typically, the first programs presented to novices contain literals. Then constants can be introduced by naming the literals, and the next step usually involves the introduction of variables. This is where the roles come in. *Fixed value* may well be presented first because it relates to the notion of constant: it is basically a constant that is set at run time, e.g., obtained as input. The next role to be introduced could be either *organizer* or *most-recent holder* but the only reasonable choice is *most-recent holder* since in introducing an *organizer* the concept of an array or some other data structure is needed. Having introduced the *most-recent holder*, "a repetitive fixed value", any other role having a relationship with the *most-recent holder* can be introduced.

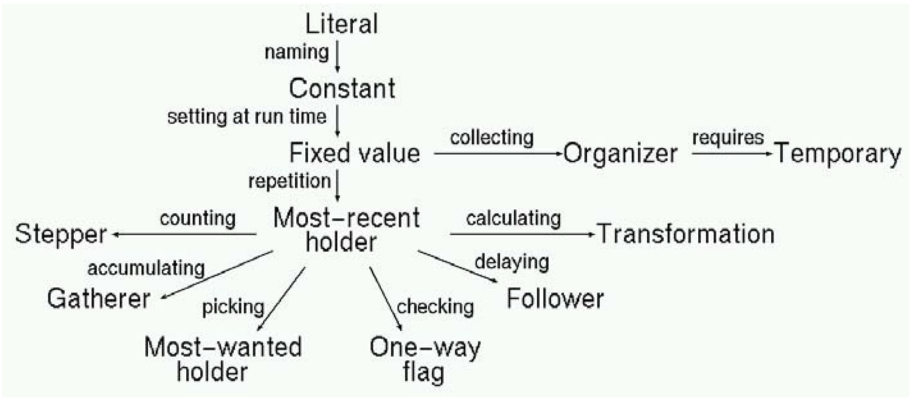
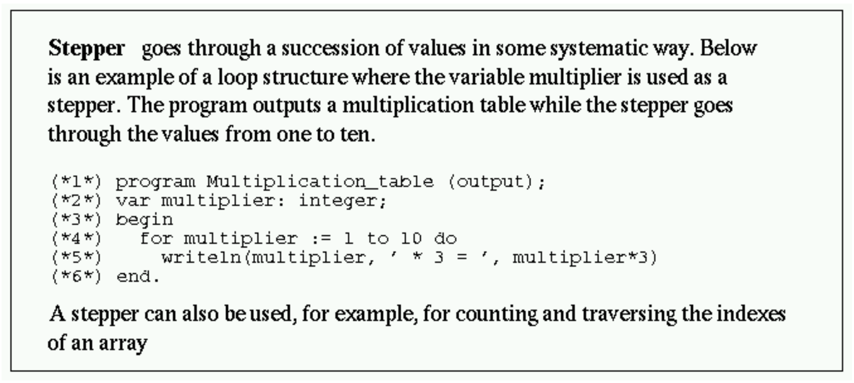


Figure 4: Relationships that can be used as a basis for incremental knowledge construction. Literal and constant are programming language constructs; other nodes are the roles (Kuittinen & Sajaniemi 2004)

A role should be introduced by giving its informal definition (see Table 1) together with additional examples of use. Moreover, the teacher may mention special cases covered by the exact definition of the role. It is essential that the distinctive features of the new role—as compared to the already known roles—are made clear. In our lectures, we have given students a printed list describing all roles in four pages. For example, we have used the description of Figure 5 accompanied with the role image in Figure 6(b) to introduce the *stepper* role in our lectures. With this technique, lectures can be based on existing materials that need only a minor update to include the role descriptions.



Stepper goes through a succession of values in some systematic way. Below is an example of a loop structure where the variable multiplier is used as a stepper. The program outputs a multiplication table while the stepper goes through the values from one to ten.

```
program Multiplication_table (output);
var multiplier: integer;
begin
  for multiplier := 1 to 10 do
    writeln(multiplier, ' * 3 = ', multiplier*3)
end.
```

A stepper can also be used, for example, for counting and traversing the indexes of an array

Figure 5: Example of a role description given to students

Role Knowledge Consolidation

Memory elaboration involves embellishing a to-be-remembered item with meaningful additional information (Anderson 2000, p. 190). More meaningful processing of learning materials results in better memories than plain repetition of the original information or addition of non-meaningful information. For example, one might repeat the informal definition of a role whenever the role reappears in some new program but this may have only a minor effect on recall. It is more important to explain how that specific variable expresses the role behavior since this is a meaningful new example of the role.

Repeated occurrences of role names in new meaningful contexts elaborate students' memory. For example, we have augmented variable declarations with role information, e.g.:

```
var closest: integer;  (* most-wanted holder,
            closest point to the center *)
```

In addition to meaningful repetition, such comments provide meaningful new examples of role behavior, e.g., a *most-wanted holder* need not be a maximum value but it can be a minimum value etc.

Another example of memory elaboration is the possibility to discuss with students about alternative role assignments. As noted in Section 3, roles are a cognitive concept and different people may assign different roles to the same variable. Therefore, it is useful to ponder how a certain variable fits the definition of a *gatherer* and at the same time the definition of a *stepper* as in the Fibonacci example in Section 3.

We have also created special role images for each role. These act as metaphors and provide new meaningful visual representations of roles' inherent properties. For example, the role image for a *stepper* in Figure 6(b) displays past and future values of the variable and makes it clear that the future values are known beforehand—an inherent property of a *stepper*.

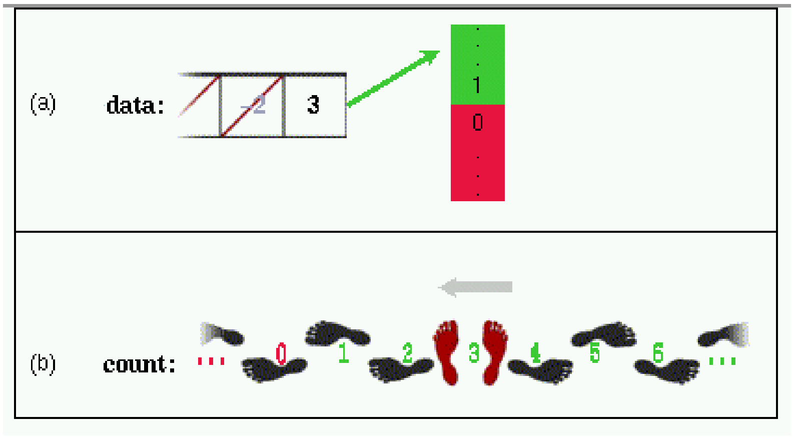
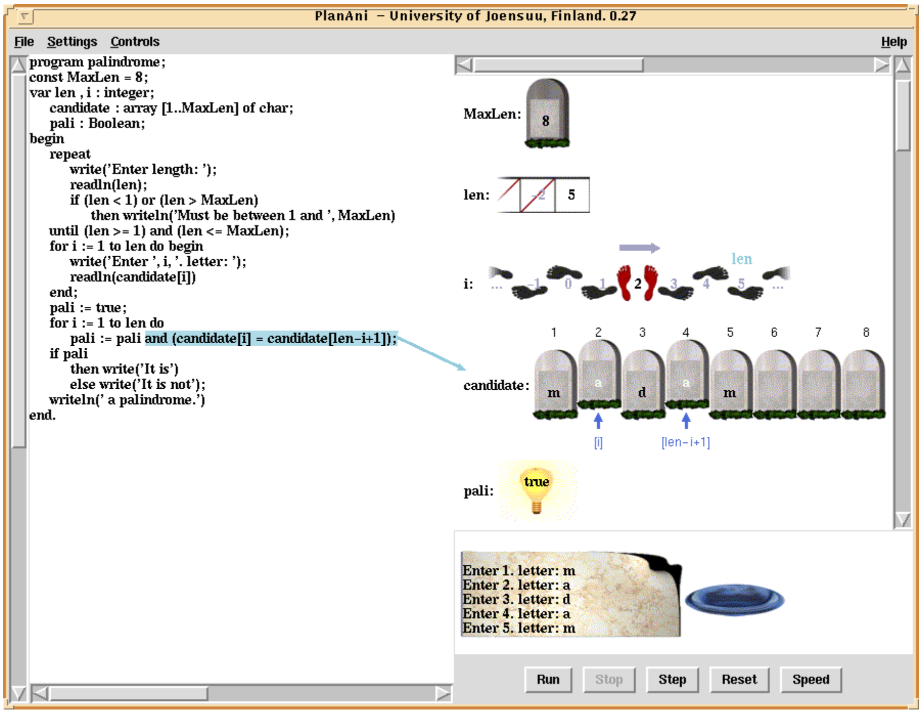


Figure 6: Visualizations of the same operation for different roles: comparing whether a *most-recent holder* (a) or a *stepper* (b) is positive (Sajaniemi & Kuittinen 2003)

We have utilized this idea in developing a program animation system, PlanAni (Sajaniemi & Kuittinen 2003). In PlanAni, a role image is used for all variables having that role. PlanAni utilizes role information for role-based animation of operations, also. For example, Figure 6 gives visualizations for two syntactically similar comparisons "some_variable>0". In case (a), the variable is a *most-recent holder* and conceptually the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, allowed values with a green background and disallowed values with red. The arrow that points to that part of the values where the current value of the variable lays, appears as green or red depending on the values it points to. In case (b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these values are now part of the variable visualization and no new values do appear. The conceptual justification is that for a *stepper* the comparison is just a check whether the end of the known sequence of values has been reached. Just like comparisons are animated differently for different roles, the animation of assignment statements depends on roles. Role-based animation of operations provides a meaningful visual representation of role properties and thus provides additional memory elaboration.

Figure 7 is an actual screenshot of the PlanAni user interface when the system is animating a simple program that checks whether its input is a palindrome. The left pane shows the animated program with a color enhancement pointing out the current action and the associated variables in the upper part of the right pane. The input/output area consists of a paper for output and a plate for input. To avoid unnecessary details PlanAni does not animate the evaluation of expressions: only the resulting value—accompanied by the expression itself—is shown and its effect in a comparison or assignment is animated.



```
program palindrome;
const MaxLen = 8;
var len , i : integer;
   candidate : array [1..MaxLen] of char;
   pali : Boolean;
begin
   repeat
      write('Enter length: ');
      readln(len);
      if (len < 1) or (len > MaxLen)
         then writeln('Must be between 1 and ', MaxLen)
   until (len >= 1) and (len <= MaxLen);
   for i := 1 to len do begin
      write('Enter ', i, '. letter: ');
      readln(candidate[i])
   end;
   pali := true;
   for i := 1 to len do
      pali := pali and (candidate[i] = candidate[len–i+1]);
   if pali
      then write('It is')
      else write('It is not');
   writeln(' a palindrome.')
end.
```

```
Enter 1. letter: m
Enter 2. letter: a
Enter 3. letter: d
Enter 4. letter: a
Enter 5. letter: m
```

Figure 7: Visualization of an array element comparison in the PlanAni system

In order to minimize users' needs to jump back and forth between the program code and the variables, the system uses frequent pop-ups that explain what is going on in the program. This includes variable creation (e.g., "*creating a gatherer called sum*"), role changes ("*the variable count acts henceforth as a stepper*"), operations ("*comparing count with zero*"), and control constructs ("*entering a loop*").

When a role appears in an animation for the first time, the teacher should explain how the role image tries to visualize the most important properties of the role. Novices have little grounds to interpret visualizations in the anticipated way (Mulholland & Eisentadt 1998). For example, the appearance of past and future values in role images might give students the impression that all these values are available and could be accessed using some special syntax. The teacher should therefore stress constantly during animation sessions that only one of the values does exist in reality and that the others are shown for visualization purposes only. In our experience, this is sufficient for preventing misunderstandings.

Programming Strategies Development

Anderson (2000) suggests that problem-solving should be taught by giving abstract instruction and concrete examples. In the programming context, this means that programming strategies should be taught by an abstract introduction of the main steps in authoring a program supported by concrete examples of program construction. In an elementary programming course the specification of a program, i.e., what the program is supposed to do, is usually given in the task assignment. Thus the first step in authoring a program is the design of the overall structure of the program. In our experience, however, novices do not know where to start. To overcome such problems, students can be taught to use data requirements and roles as a starting point in program design.

As a concrete example, consider the task of writing a program to convert temperatures between various scales when the input to the program consists of a temperature value and its unit (Celsius, Kelvin, or Fahrenheit) and the program has to output the temperature in all the three scales (Kuittinen & Sajaniemi 2004). Program design may now start with selecting variables found in the specification, i.e., input and output values. Input is normally stored in a *fixed value* (if only a single value is read) or in a *most-recent holder*; so one of those is needed for both inputs; let's call them degree and unit. The declarations of these variables do not depend on the role; the difference between the two roles is that a *most-recent holder* suggests the use of a loop. The program specification does not require repetition, so *fixed values* will suffice.

```
program ?name? (input, output);
var degree:        ?type? (* fixed value: input temperature *)
   unit:           ?type? (* fixed value: input unit        *)
   degCelsius:     ?type? (* transformation: input in C     *)
   degKelvin:      ?type? (* transformation: input in K     *)
   degFahrenheit:  ?type? (* transformation: input in F     *)
begin
   input degree
   input unit
   degCelsius := ?calculation based on input degree?
   degKelvin := ?calculation based on input degree?
   degFahrenheit := ?calculation based on input degree?
   output all degrees
end.
```

Figure 8: First draft of a program obtained by analyzing data requirements in the program specification (Kuittinen & Sajaniemi 2004)

The program has to produce the same temperature in several units. These are *transformations* of the original value, so we will need one for each. The program looks now as in Figure 8. This program utilizes two roles and their prototypical uses in programs. It does not say what should be written between question marks but it gives some hints on the kind of things there might be and provides a concrete base to build on. There is no guarantee that the method suggested above would produce an optimal program. However, it gives a possibility to start program design directly on the basis of the problem statement. Thus, roles can be used to teach elementary programming strategies.

The example above is actually based on real life: one fourth-year student once told me about his experiences in helping a novice to do a home assignment. He had had problems in getting the novice to begin writing the program: the novice just couldn't know where to start. The novice's problem was something like "Do I need a variable or a loop? ", that is, the novice could not see how various programming constructs are used in the different steps of program creation. Neither of these students had learned the role concept and they ended up with a solution different from that of Figure 8.

OTHER RESEARCH INTO ROLES OF VARIABLES

The previous sections have described the role concept, how it can be utilized in teaching elementary programming, and what effects of the use of roles in teaching have been discovered. This section surveys other research on roles: roles in expert programmers' knowledge, computer science educators' attitudes to roles, and roles in various programming paradigms.

Roles and Experts' Programming Knowledge

The individual roles were originally identified by studying all variables in three textbooks and by creating a classification for them (Sajaniemi 2002). Thus it is possible that roles are artificial concepts with no relationship to experts' programming knowledge. In order to study this more carefully, we conducted a knowledge elicitation investigation where professional programmers studied programs and the resulting mental representations were elicited (Sajaniemi & Navarro Prieto 2005).

Participants of this study consisted of thirteen expert programmers with an average background of 13.7 years of professional programming. A participant's task was to study five short C programs and to modify each program (in order to make sure that he understood the programs well). The researcher then gave cards representing the variables in all programs and asked the participant to sort the cards in groups so that "similar variables will go together". When the sorting task was ready, the participant was asked to give a written explanation for each of his groups. This was followed by an interview where the participant explained the sorting criterion he had used, the exact contents of each group, and alternative sorting criteria he had thought of or might consider to be appropriate.

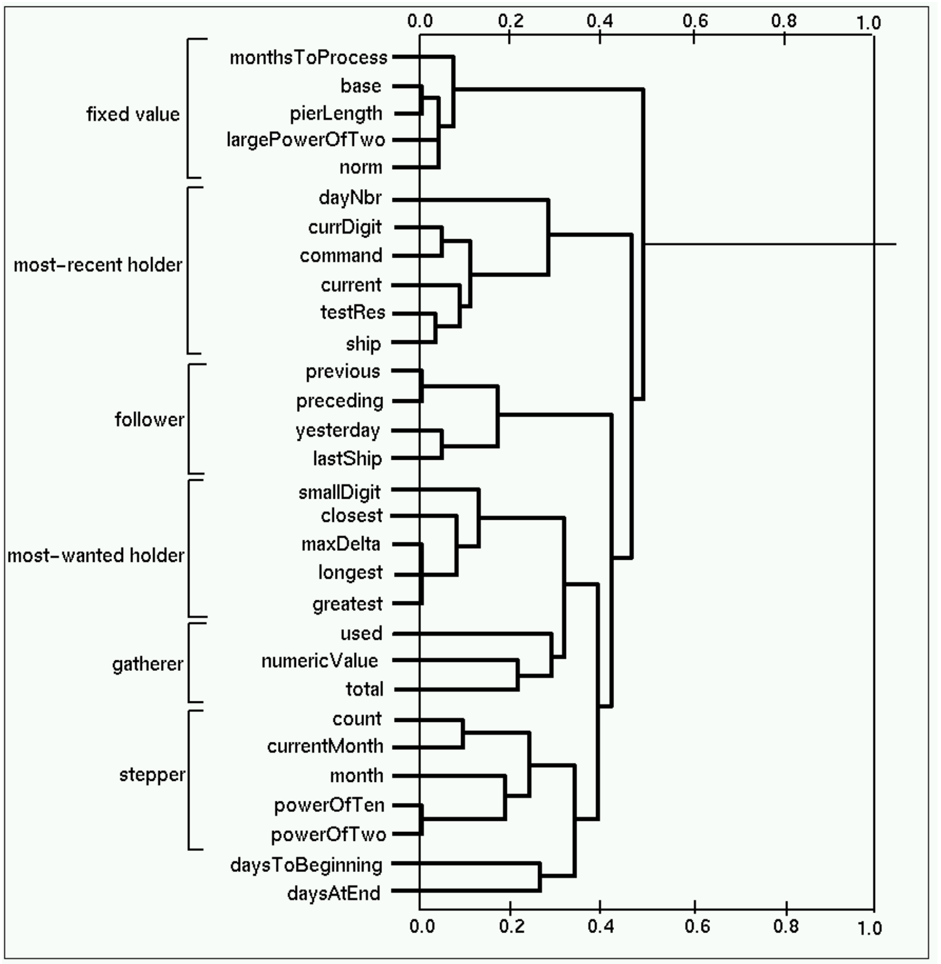


Figure 9: Result dendrogram of the hierarchical cluster analysis (Sajaniemi & Navarro Prieto 2005)

The experts used or mentioned fourteen different criterion principles that may be organized in four main categories. Domain-based criteria deal with issues related to the application domain, technology-based criteria deal with the features of programming languages, execution-based criteria are based on activities that occur during the execution of a program, and strategy-based criteria have their origins in various programming-related strategies. Each criterion principle may give rise to several sorting criteria with differences among details. Even though behavior, i.e., role-like sorting criteria was only one of the fourteen principles, it had a dominant effect on the groups formed by the participants. Figure 9 depicts the results of a hierarchical cluster analysis that produces a general view of the groups. Most of the variables are clustered according to the roles defined in Section 3. The last two variables—daysToBeginning and daysAtEnd—were supposed to be a follower and a gatherer, respectively. However, they allow several interpretations in the role theory and therefore it is only natural that different experts put them into different groups.

The analysis of the interviews revealed that many experts based their sorting at least partly on the behavior of variables, i.e., on role-like properties. Moreover, all the roles included in the experimental materials were identified in the groups. Thus roles were found to be a part of experts' programming knowledge.

However, there appears to be two sources of variation in the judgment of roles: what behavior do programmers perceive from the lifetime of a variable, and what behaviors are considered to be similar. The behavior of a variable may be perceived differently by two persons even though they look at the same variable, at the same operations on the variable, and at the same value sequence. As a result, they will perceive a different role for the same variable. An example of this type of variation was given in Section 3 where two different interpretations of the Fibonacci sequence were presented.

As an example of the second source of variation consider repeated addition by a constant on one hand, and repeated division by a constant on the other hand. Some experts considered these to be similar behaviors (and thus sorted them together in a group corresponding to the role *stepper*), others considered these to be two different behaviors, and some were unsure ("these are kind-of counters"). This variation is manifested in vague role boundaries and in differences in the granularity of the roles.

The experiment thus proved that roles are a part of expert programmers' tacit knowledge and explained why people disagree on a role in some cases, i.e., found two different reasons for the cognitive nature of the role concept.

Roles and CS Educators

In order to study the understandability and acceptability of the role concept and of the individual roles as seen by CS educators we conducted a web-based investigation (Ben-Ari & Sajaniemi 2004). The research materials consisted of web pages divided into three phases. The tutorial phase introduced the role concept and contained descriptions of individual roles; in the training phase participants had to recognize roles of variables in short programs and they got feedback on their assignments of roles; the analysis phase was similar in format to the training phase but no feedback was given and the results were sent by email to us. The roles of *one-way flag*, *temporary* and *organizer* accounting for only 5.2 % of all variables in the analysis of textbooks (Sajaniemi 2002) were not included in the materials in order to simplify and shorten the overall task. As a result, the participants reported using 45-90 minutes to the whole task.

Table 2: Participants' selections for the roles (percent) (Ben-Ari & Sajaniemi 2004)

Role	**Role selected**						
	FIX	**STP**	**MRH**	**MWH**	**GTH**	**TRN**	**FOL**
FIX	**91**		7		2		
STP		**91**	2		1	4	2
MRH	7	1	**92**				
MWH		1	1	**79**	3	3	10
GTH	1	1	10	1	**60**	26	1
TRN	9	1	7	3	1	**75**	4
FOL			2				**96**

Table 2 displays the selections made by the participants for each role in the analysis phase. Most roles were identified by at least 90 % accuracy. The low success in identifying of *most-wanted holders* and *gatherers* can be explained by controversial variables, i.e., cases where several alternative interpretations of the role are possible. In non-controversial cases, *most-wanted holders* were identified correctly in 91 % of the cases, and *gatherers* in 94 % of the cases. *Transformations* were recognized with 75 % accuracy, though even here, in a simple case the role was recognized with 90 % accuracy.

Many participants stated in their comments that they had had problems in remembering the definitions of the roles or that the definitions were ambiguous. However, the same participants scored between zero to two errors on non-controversial variables, indicating that they understood, perhaps subconsciously, the deep structure of variables represented by the roles. CS educators' comments on the role concept in general were mostly positive, and they believed that roles could contribute to understanding programs.

Roles in Different Programming Paradigms

The previous subsection described an investigation into the understandability and acceptability of the role concept and of the individual roles in procedural programming. We have conducted a similar investigation also in object-oriented programming (Byckling et al. 2005) and in functional programming (Kulikova 2005). Both of these investigations were preceded by an analysis of variable use in elementary programming textbooks of the paradigm in question.

In object-oriented programming, novice-level programs appear to be more complicated than in procedural programming and some new roles are therefore needed. For example, linked structures are usually introduced in the first OO course but later when teaching procedural programming. Linked structures seem to need a new role "walker", which goes through a data structure; in object-oriented programming this kind of role is typically called an iterator. We have not studied in detail what new roles are really needed, but they seem to concern data structures and will probably be needed in procedural programming, also. On the other hand, all the present roles apply to object-oriented programming and they were identified in object-oriented textbooks. Thus there is no need to alter present roles for the purposes of object-orientation.

In larger programs it can be hard for a human to extract the behavior of a variable when the lines affecting its value are dispersed. In object-oriented programming, data flow is decentralized and all lines affecting a variable are not necessarily in the same method—in fact they can be in different classes—and control flow is harder to reveal than in procedural programming (Corritore & Wiedenbeck 1999). As a result, participants of the OO web survey had problems in assigning roles to some variables. This does not, however, mean that roles would be less important in object-oriented programming. On the contrary, explicit role information in the form of comments written by the author of a program might help expert programmers in program comprehension; knowing that a variable is, e.g., a *stepper* indicates the succession of values it may obtain.

In OO, roles can be assigned not only to variables inside methods but also to attributes in classes. Moreover, some objects can also have a “variable-like” behavior. In Java, e.g., a String-type or an Integer-type object capsulates just one value. All the events directed to the object will have an effect on the value of the only attribute and the whole object behaves just like a primitive variable. In such cases, the object is considered to have the role of its only attribute.

```
fun max(a, nil)     = a
|   max(a, (h::t))  = if h>a then max(h,t)
                      else max(a,t)
```

Figure 10: Finding maximum in functional programming

In functional programming there are no variables. However, function parameters as well as return values of recursive functions have role-like behavior. For example, consider the function max in Figure 10 that finds the maximum in a list of values. During recursive calls, the parameter a is always the largest value found so far, i.e., a *most-wanted holder*, the parameter h is the current value, i.e., a *most-recent holder* etc. Thus the set of entities that have roles is different in different programming paradigms: variables in procedural programming; parameters and return values in functional programming; variables, attributes and (some) objects in object-oriented programming.

In the analysis of functional programming textbooks (Kulikova 2005), no *temporaries* were encountered. A similar effect can be found also in other paradigms if a temporary variable is declared locally in the block that needs it. Theoretically this makes the variable a *fixed value*. Kulikova (2005) also suggests two new roles for functional programming: *modifier* which is a data structure that allows modifications, and *selector* which is a special case of *most-wanted holder*. Again, these new roles are related to data structures and are most probably needed in intermediate level procedural and object-oriented programming, also.

CONCLUSION

Roles of variables were first introduced in 2002 and we have studied them in detail since then. In this talk, I have presented our results concerning several aspects of the roles: their detailed properties, their applicability to different programming paradigms, their psychological existence among programmers' tacit knowledge, their understandability from educators point of view, and their effects on learning programming. In addition to the work reported above, we have studied several aspects of the role-based program animator PlanAni in more detail; as well as automatic role detection that could provide help for program comprehension in professional maintenance tasks. The work is not yet finished; we are currently continuing our work to understand the role concept more thoroughly in the hope of providing better concepts and tools for both the learner and the professional programmer.

ACKNOWLEDGMENTS

I would like to thank all the people contributing to this work: my co-researchers Marja Kuittinen, Mordechai Ben-Ari, and Raquel Navarro Prieto; doctoral students Pauli Byckling, Petri Gerdt, Seppo Nevalainen, and Tuija Stützle; many undergraduate students that have made their theses or special projects in this research project; and the numerous volunteers participating in the experiments.
This work was supported by the Academy of Finland under grant number 206574.

REFERENCES

Anderson, J. R. (2000), *Cognitive Psychology and its Implications,* Worth Publishers (5th edition)

Ben-Ari, M. (2001), Constructivism in computer science education, *Journal of Computers in Mathematics and Science Teaching*, 20(1), 45-73

Ben-Ari, M. & Sajaniemi, J. (2004), Roles of variables as seen by CS educators, In *Proceedings of the Ninth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)*, 52-56, ACM Press

Ben-David Kolikant, Y. & Haberman, B. (2001), Activating "black boxes" instead of opening "zippers" - a method of teaching novices, In *Proceedings of the Sixth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*, 41-44, ACM Press

Bereiter, C. (1994), Constructivism, socioculturalism and Popper's world, *Educational Researcher*, 23(7), 21-23

Byckling, P., Gerdt, P. & Sajaniemi, J. (2005), Roles of variables in object-oriented programming, Accepted to the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005) Educators Symposium, San Diego, California, USA, October 2005

Byckling, P. & Sajaniemi, J. (2005), Using roles of variables in teaching: Effects on program construction, In P. Romero, J. Good, S. Bryant, & E. A. Chaparro (Eds.), *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, 278-303, University of Sussex, UK

Clancy, M. J. & Linn, M. C. (1999), Patterns and pedagogy, In *Proceedings of the 30th SIGCSE Technical Symposium on CS Education*, *ACM SIGCSE Bulletin*, 31(1), 37-42

Corritore, C. & Wiedenbeck, S. (1999), Mental representations of expert procedural and object-oriented programmers in a software maintenance task, *International Journal of Human-Computer Studies*, 50, 61-83

Ehrlich, K. & Soloway, E. (1984), An empirical investigation of the tacit plan knowledge in programming, In J. C. Thomas & M. L. Schneider (Eds), *Human Factors in Computer Systems*, 113-133, Norwood, NJ: Ablex Publishing Company

Good, J. & Brna, P. (2004), Program comprehension and authentic measurement: A scheme for analysing descriptions of programs, *International Journal of Human-Computer Studies*, 61(2), 169-185

Green, T. R. G. & Cornah, A. J. (1985), The programmer's torch, In *Human-Computer Interaction - INTERACT'84*, 397-402, IFIP, Elsevier Science Publishers

Holland, S., Griffiths, R. & Woodman, M. (1997), Avoiding object misconceptions, In *Proceedings of the 28th SIGCSE Technical Symposium on CS Education*, *ACM SIGCSE* Bulletin, 29(1), 131-134

Kuittinen, M. & Sajaniemi, J. (2004), Teaching roles of variables in elementary programming course, In *Proceedings of the Ninth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'04)*, 57–61, ACM Press

Kulikova, Y. (2005), Roles of variables in functional programming, Unpublished Master's Thesis, Department of Computer Science, University of Joensuu, Finland

Mulholland, P. & Eisenstadt, M. (1998), Using software to teach programming: Past, present and future, In J. Stasko, J. Domingue, M. H. Brown, and B. A. Price (eds), *Software Visualization - Programming as a Multimedia Experience*, 399-408, The MIT Press

Nguyen, D. (1998), Design patterns for data structures, In *Proceedings of the 29th SIGCSE Technical Symposium on CS Education, ACM SIGCSE Bulletin*, 30(1), 336-340

Pennington, N. (1987), Comprehension strategies in programming, In G. M. Olson, S. Sheppard and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*, 100-113, Norwood, NJ: Ablex Publishing Company

Petre, M. & Blackwell, A. F. (1999), Mental imagery in program design and visual programming, *International Journal of Human-Computer Studies*, 51(1), 7-30

Rist, R. S. (1989), Schema creation in programming, *Cognitive Science*, 13, 389-414

Rist, R. S. (1991), Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers, *Human-Computer Interaction*, 6, 1-46

Sajaniemi, J. (2002), An empirical analysis of roles of variables in novice-level procedural programs, In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 37-39, IEEE Computer Society

Sajaniemi, J. (2005), Roles of variables home page, http://www.cs.joensuu.fi/saja/var_roles

Sajaniemi, J. & Kuittinen, M. (2003), Program animation based on the roles of variables, In *Proceedings of ACM 2003 Symposium on Software Visualization (SoftVis 2003)*, 7-16, ACM Press

Sajaniemi, J. & Kuittinen, M. (2005), An experiment on using roles of variables in teaching introductory programming, *Computer Science Education*, 15, 59-82

Sajaniemi, J. & Navarro Prieto, R. (2005), Roles of variables in experts' programming knowledge, In P. Romero, J. Good, S. Bryant & E. A. Chaparro (Eds.), *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005)*, 145-159, University of Sussex, UK

Samurçay, R. (1989), The concept of variable in programming: Its meaning and use in problem-solving by novice programmers, In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer*, 161-178, Hillsdale, NJ: Lawrence Erlbaum Associates

Sleeman, D., Putnam, R. T., Baxter, J. A. & Kuspa, L. (1989), A summary of misconceptions of high school Basic programmers, In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer*, 161-178, Hillsdale, NJ: Lawrence Erlbaum Associates

von Glasersfeld, E. (1995), A constructivist approach to teaching, In P. Steffe & J. Gale (Eds.), *Constructivism in Education*, 3-15, Hillsdale, NJ: Lawrence Erlbaum Associates

Wallingford, E. (2003), The elementary patterns home page, http://www.cs.uni.edu/wallingf/patterns/elementary