Teaching Novice Programming Using Goals and Plans in a Visual Notation

Minjie Hu  
Department of Information Science  
University of Otago  
PO Box 56, Dunedin 9054  
minjiehu@infoscience.otago.ac.nz  
Taiarawhiti Campus  
Eastern Institute of Technology  
PO Box 640, Gisborne 4010  
New Zealand  
mhu@eit.ac.nz

Michael Winikoff  
Department of Information Science  
University of Otago  
PO Box 56, Dunedin 9054  
New Zealand  
mwinikoff@infoscience.otago.ac.nz

Stephen Cranefield  
Department of Information Science  
University of Otago  
PO Box 56, Dunedin 9054  
New Zealand  
s cranefield@infoscience.otago.ac.nz

Abstract

Introductory programming courses have been continuously reported as having a high rate of failure or withdrawal. This research aims to develop a new approach for teaching novice programming, which is both easy to introduce and effective in improving novice learning. Our approach combines three key ideas: using a visual programming language; using strategies, specifically using the concepts of “goal” and “plan”; and having a well-defined process. We present a way of representing goals and plans in a visual notation together with a plan library that we developed in a visual programming environment (VPE). A key feature of the approach is that a design, i.e. an unmerged “plan network”, is executable and can be tested, giving feedback in the VPE. Furthermore, we describe a detailed process for using existing plans and building new plans in the VPE. This approach had been evaluated experimentally and the results indicated its potential to significantly improve teaching programming to novices.

Keywords: Goal, Plan, Visual Notation, Process of Programming.

1 Introduction

Although a wide range of approaches have been proposed to improve novices’ learning of programming (Kay et al. 2000, Pears et al. 2007, Robins, Rountree, and Rountree 2003, Rößling et al. 2008), there continues to be a high rate of failing or withdrawing from the first programming course (Lahtinen, Ala-Mutka, and Järvinen 2005, Sykes 2007). A number of reasons have been proposed for this, such as “fragile” knowledge of programming concepts (Lister et al. 2004, McCracken et al. 2001), lack of problem-solving strategies and plans (de Raadt 2008, Winslow 1996), and lack of detailed mental models (du Boulay1989, Winslow 1996). There seems to be a broad consensus that “novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs” (Winslow 1996, page 17). This research therefore focuses on the heart of the matter: how to teach novices to construct programs.

The basis for this work is the hypothesis that what is needed is a process that students can follow, along with a structured means of representing the parts of a solution using an easy-to-use notation. Specifically, we conjecture that combining goals and plans with a detailed process will yield an effective means for teaching programming. Our proposed approach thus combines three ideas: using a Visual Programming Environment (VPE), using goals and plans, and having a well defined process. This combination is novel and carefully motivated. We have chosen to use a VPE (Kelleher and Pausch 2005) (specifically Scratch1 (Resnick et al. 2009)) since VPEs aim to provide an attractive, easy, and fun way for novices to learn programming. In VPEs, such as Alice (Sykes 2007) and Scratch, programs are built by dragging and dropping statement blocks, which helps to prevent syntax errors and avoids the need to learn and memorize syntax. The idea of goals and plans is based on the finding that experts use strategies to solve programming problems (Soloway 1986). We follow other researchers (de Raadt 2008, Guzdial et al. 1998) in using goals and plans. We represent them explicitly, by devising a visual notation for goals and plans, and extending the Scratch language with an explicit representation for plans. Finally, we define a detailed process to guide novices through the activities of programming. This process is not just a high-level sequence of steps, but includes detailed guidance for how to perform sub-steps in this process.

The remainder of this paper is structured as follows. In Section 2, we discuss related work, in particular work that uses the goal and plan concepts and a programming process. Section 3 presents the explicit representation of goals and plans. In Section 4, a well defined programming process utilising an existing plan library is described, and in Section 5, a process for building new plans is given. Section 6 presents an evaluation of the proposed approach. Finally, we conclude in Section 7.

2 Literature Review

A goal is a certain objective that a program must achieve in order to solve a problem (Letovsky and Soloway

1 http://scratch.mit.edu
1986), and a plan (Spohrer, Soloway, and Pope 1985) corresponds to a fragment of code that performs actions to achieve a goal. Goals and plans are key components in representing problems and solutions (Soloway 1986).

In the 1980s, Soloway and his colleagues (Letovsky and Soloway 1986, Soloway 1986, Spohrer et al. 1985) discovered that experts have strategies to solve problems using their library of plans. They advocated teaching these strategies and plans explicitly so that novices could have sufficient instructions on how to “put the pieces together”. Concurrently, they proposed (Soloway 1986) to use a goal and plan “language” for novices to explicitly construct their own plans. Moreover, a tool, GP(ducer) (Guzdial et al. 1998) was created that supported novices to write a program based on the decomposition and composition of goals and plans. However, there was not a detailed process to support the composition of pieces of plan code, and, furthermore, the tool’s evaluation did not clearly demonstrate a significant advantage.

Similarly, pedagogical programming patterns were advocated by Porter and Calder (2003) by using small programming pieces in teaching novice programmers. Once again, a tool, ProPAT, was inspired by the idea of programming patterns (de Barros et al. 2005) allowing novices to insert code from the pedagogical patterns. However, there was weak support for how to apply these patterns in the goal analysis. Furthermore, there was not a detailed process to support the composition of pieces of plan code, and, additionally, the tool’s evaluation did not clearly demonstrate a significant advantage.

Recently, the goal and plan concepts have been taught as programming strategies in curricula by de Raadt (2008). Each strategy was also called a plan, which was basically pattern-like program code with examples. This approach attempted to integrate plans to build the program code after explicitly introducing goals and plans. However, it lacked a detailed process of programming development from goals to program via plans. Firstly, there were no clear guidelines for performing goal analysis. Secondly, there was no well defined process for merging plans. Thirdly, debugging and testing were excluded from the strategies in the program implementation.

As we have seen, a number of approaches that have used goals and plans have failed to provide detailed processes. On the other hand, there are approaches that have provided detailed processes, but they tended not to use the concepts of goals and plans. For example, “Programming by Numbers” (Glaser, Hartel, and Garratt 2000) provides a clear process to create the smallest components of functions. It breaks the programming process into a series of well-defined steps and gives students a way of “programming in the small”. A similar, but more detailed, systematic design method was applied by Felleisen et al. (2004) to produce well-specified intermediate products in a stepwise fashion called “TeachScheme”. However, although both approaches emphasize the detailed process, the goal and plan concepts are not included. Additionally, both approaches are data-driven and more suited to functional programming languages than to mainstream procedural languages. More recently, a stepwise improvement process, STREAM (Caspersen and Kölling 2009), was developed as a conceptual framework with six major steps for teaching novice object-oriented (OO) programming together with five rules for implementing OO methods in order to break the task into smaller steps.

Table 1 summarizes the related work in terms of whether it uses goals and plans, whether a detailed process is provided, and whether testing and debugging are supported. As can be seen, existing work tends to either use goals and plans, but not provide a detailed process for guiding novices; or it provides a detailed process but does not use goals and plans.

<table>
<thead>
<tr>
<th>Literature Study</th>
<th>Goals</th>
<th>Plans /Patterns / Schemata</th>
<th>Detailed Process</th>
<th>Test /Debug</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soloway and his colleagues (1980s &amp; 90s)</td>
<td>Yes</td>
<td>Plans in Code</td>
<td>Weak</td>
<td>N/A</td>
<td>Pascal</td>
</tr>
<tr>
<td>de Barros et al. (2005)</td>
<td>Weak</td>
<td>Patterns in Code</td>
<td>Weak</td>
<td>Weak</td>
<td>C</td>
</tr>
<tr>
<td>Bielikova and Navrat (1998)</td>
<td>Weak</td>
<td>Schemata</td>
<td>Weak</td>
<td>No</td>
<td>Prolog</td>
</tr>
<tr>
<td>Glaser et al. (2000)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>ML</td>
</tr>
<tr>
<td>Felleisen et al. (2004)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Scheme</td>
</tr>
<tr>
<td>Caspersen and Kölling 2009</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Java</td>
</tr>
<tr>
<td><strong>This Research</strong></td>
<td>Yes</td>
<td>Plans in VPE</td>
<td>Yes</td>
<td>Yes</td>
<td>Visual Programming Language</td>
</tr>
</tbody>
</table>

3 Representing Goals and Plans

In order to be able to develop designs in terms of goals and plans, we need to have a way of representing them. Using a VPE such as Scratch, it is thus crucial to develop a visual notation for goals and plans.

3.1 The Visual Notation of Goals

Every program has a certain number of goals to be accomplished. Our visual notation for goals distinguishes between three basic types of goals: Input, Process, and Output (see Figure 1). A simple program might have one goal of each type, and achieve these goals in sequence. More generally, a program may have multiple goals of each type, and these goals can be combined using a mixture of sequential and parallel composition. This order is indicated graphically (see Figure 2), where the order of goal processing is left-to-right, and where goal decomposition is indicated by nesting. For example, Figure 2 shows that Goal 1 (an input goal) is followed by an unnamed goal which has been decomposed into three processing sub-goals: Goal 2 and 3 (which are achieved in...
parallel), followed by Goal 4. These three goals are then followed by Goal 5 (an output goal).

3.2 The Visual Notation of Plans

The visual plans are built up in order to implement the blueprint of the goals, following the metaphor of a network of plans that communicate using dataflow.

Each goal corresponds to a plan. Accordingly, there are three types of plans: input plans, process plans, and output plans. An input plan inputs data and then produces an output dataflow. Conversely, an output plan consumes the dataflow and displays it. A process plan consumes its input dataflow, and then processes it to produce a new dataflow in order to achieve the goal of the process.

Each plan is represented by a plan block. Plan blocks (see Figure 3) are constructed using BYOB\(^2\), an extension to Scratch that allows you to Build Your Own Blocks. Each plan block has a unique name and parameter(s), including named “plan ports” (either “in” or “out”), which are used to connect the dataflow between the plans. A plan block also has an internal definition (not shown in Figure 4) which is just a BYOB process, constructed from standard BYOB constructs and the provided scaffolding blocks (discussed below). A group of existing plan blocks are developed and called a visual plan library. The visual plan library supports novices in designing and implementing their program (see Figure 4).

A set of visual plans can be linked by their dataflow ports to achieve a given set of goals (see step 2 in Figure 6). The set of plans is viewed as a network where items of data “flow” from one plan to another. The advantage of this “plan network” model is that because plans are conceptually concurrent, there is no need to worry about the correct sequencing of plans. This allows the plan network to be executable even before the plans have been merged (see Section 4.2).

In order to allow the plan network to be executable we have developed some special blocks which send and receive data to and from other plan blocks, and which indicate the linkage between the plan blocks. These blocks are referred to as “scaffolding blocks” and they are used to define the linkages between plan blocks (and also to send and receive data between plan blocks – see Section 5). For example, a “Begin Links” scaffolding block (See Figure 5 and step 2 in Figure 6) is followed by a number of “Link” scaffolding blocks, which capture the links between the plan blocks. Each “Link” block links two plans by its two parameters. The first parameter indicates the out-port of a plan and the second parameter links this to the in-port of another plan. An “End Links” block indicates the end of the collection of “Link” blocks. A further example of using scaffolding blocks in the process of converting the plan-based solution into a single BYOB program will be illustrated in Section 4. The method of applying scaffolding blocks to construct new plan blocks will be described in Section 5.

4 The Process of Programming

The process of programming from the visual notation of goals and plans to the final program consists of five steps: (1) analysing goals; (2) designing a network of plan blocks; (3) expanding the plan blocks; (4) merging the expanded plan details; and (5) simplifying the merged details (see Figure 6). The process is illustrated using the following example, which was also used by both Soloway (1986) and de Raadt (2008) to analyse goals and plans:

Write a program that will read in integers and output their average. Stop reading when the value -1 is input.

---

\(^2\) http://byob.berkeley.edu
Figure 6. The process of programming from visual notation of goals and plans to the final program
4.1 Analysing Goals

Analysing goals involves identifying what goals the program needs to achieve. Typically these include at least one input, one output, and some number of processing goals, some of which may have sub-goals.

Identifying goals is done by a process of problem decomposition. For example, in order to achieve the goal of displaying the average of a sequence of values, one needs to read input, compute the average, and then display it. These correspond to goals. The second goal, computing the average, can be decomposed further resulting in five goals (see step 1 in Figure 6). The first goal is to input the values to be averaged. The program then needs to obtain both a “sum” and “count” from these values. It then accomplishes the goal of dividing the “sum” by the “count” giving the valid result “average”. Finally, the result will be displayed. Note that the goal notation is used purely as a design aid: we have not (yet) extended the BYOB tool to provide support for the goal analysis step.

4.2 Designing a Network of Plan Blocks

The network of plans is derived from the goal analysis by creating a plan for each goal, and deriving the dataflow based on the data requirements, in accordance with the goal ordering. For example, the dataflow starts from the “input” goal. The “sum” goal is after the “input” goal, and therefore data can flow from the “input” plan to the “sum” plan. Meanwhile data can also flow from the “input” plan to the “count” plan. After both goals of “sum” and “count” are achieved by the corresponding plans, the goal of “average” can be completed by a “dividing” plan, consuming dataflow from both the “sum” and “count” plans. Finally, the goal of displaying the “average” is reached by the “output” plan according to the dataflow from the “dividing” plan.

Based on the goal analysis, five plan blocks will be applied to build up a “plan network” in order to achieve these goals (see step 2 in Figure 6). The resulting design is captured using plan blocks and link blocks, with the plan blocks being listed in sequence, based on the ordering of the goals identified in the goal analysis (since the goals may not be in a strict sequence, the ordering is a partial one). A set of link blocks is then used to capture the links between plan blocks from out-port to in-port, based on the order of dataflow between them. For example, “Link values1:out to sum:in” links the out-port of the “input” plan to the in-port of the “sum” plan. Meanwhile, “Link values1:out to count:in” also connects the out-port of the “input” plan to the in-port of the “count” plan. Therefore, the dataflow from the “input” plan is copied and made available to both “sum” and “count”. Conversely, the dataflows from the out-ports of both “sum” and “count” plans are joined together into the “dividing” plan by the next two link blocks. Finally, the result is linked from the “dividing” plan to the “output” plan.

As noted earlier, a key feature of our approach is that this design can be tested by running the program containing the scaffolding blocks. Furthermore, the scaffolding blocks inside the plan block provide feedback if the parameters in the “Link” block do not match the names in the plan block.

4.3 Expanding the Plan Blocks

Expanding plan blocks means replacing each plan block with the defined block details within it (see step 3 in Figure 6). Whereas the previous two steps, goal analysis and designing a network of plan blocks, require human thought and creativity, this step is purely mechanical and could be automated (although adding this support to BYOB is future work).

The result of expanding the plan blocks is also executable and testable, which allows novices to obtain feedback rather than having to wait until the end of the process. Visualisation of dataflow through the plan-ports can be seen on the BYOB “Stage” by stepping the blocks to provide feedback.

4.4 Merging the Plan Details

Merging plan details aims to combine the plan details of the different plans into one program that does not make essential use of the scaffolding blocks (although they are still present; see step 4 in Figure 6). The merge principle is presented as a set of rules and also demonstrated to students using examples. Although this process is clearly defined, it is somewhat complex, and providing tool support for the merging process is a key direction for future work.

The basic steps of merging the plan details are:

1. collect all the blocks that initialise variables, and put them at the start of the program, i.e. immediately after the “End Links” block (e.g. the first two statements, “set Sum to 0” and “set Count to 0”, of the merged plan details in Figure 6, step 4);
2. next, put together the statements (including input statements) that initialise variables that are used in loop conditions (e.g. the 3rd and 4th statements, “ask” and “set”, in the merged plan details in Figure 6, step 4);
3. merge two loops when the second loop is “driven” by output from the first loop. Specifically, when the first loop outputs values via a port that is linked to an input port of a second loop, and the second loop has the structure “repeat until NO MORE DATA?(in-port)”. For example, the body of the “sum” Plan has a loop that gets data from an input port (“sum:in”) which is linked to the output port of the “input” plan (“values1:out”). In other words, after the first loop from the “input” plan sends data to the out-port “values1”, the first loop from “sum” plan gets data from the linked in-port “sum:in”. In other words the output from the first loop “drives” the second loop. Thus, the loop from the “input” plan is merged with the loop from the “sum” plan. When merging loops, the statements in each of the loops are kept in order, with the statements from the second loop being placed after those from the first loop. An exception is (input) statements that affect variables used in the loop condition: these are put at the end of the loop. For example, the two “set” statements from the second loop (“set Number to GET DATA sum:in” and “set Sum to SUM +
Number”) are put into the first loop after the original statement “SEND DATA Number values1:out”. However, the input statements “ask” and “set” affect the loop condition, and so are put at the end of the merged loop. Similarly, the loop of the “count” plan is also merged.

4. any loop which is driven by reading a dataflow that will only have a single value can be simplified by replacing the loop with its body (e.g. the fourth and the fifth loops).

Once more, the result of merged plan details is executable and testable with the values of variables being visible in the BYOB stage.

4.5 Simplifying the Merged Details

The last step in the process is to simplify the merged details. This is done by combining variables that deal with the same data but have different variable names, and then removing all the scaffolding blocks to obtain the final program. The steps are:

1. if a variable’s value is sent on an output port, and subsequently read from the linked input port into another variable, then the second variable should be consistently replaced with the first one. For example, consider the sequence “SEND DATA Sum sum:out”, and “set Number1 to GET DATA dividing:in.dividend”. Because the two ports are linked, the value of Number1 is taken from Sum, and so Number1 should be consistently replaced with Sum. Similarly, variable Number2 is replaced by variable Count;

2. remove the use of ports and the associated scaffolding blocks (e.g. blocks from “Begin Links” to “End Links”, all the “SEND DATA” blocks, and all the blocks containing block “GET DATA”). This results in the final program shown in Figure 6.

5 Build Your Own Plan Blocks

The process presented in the previous section can be used by a novice programmer to develop a program using the plan blocks in the visual plan library. After the user has become familiar with the process of programming from goals and plans by using the existing plan blocks in BYOB, the next stage is to have them build their own plan blocks to add to the plan library. Since the input and output plan blocks can be used in most situations, the following only describes (very briefly) how to build process plan blocks. These are built using Scratch’s constructs (control flow, assignment, etc.) and three scaffolding blocks (see Figure 7): NO MORE DATA? (in-port) which returns a “true” or “false” value to indicate whether there is any more data from the “in-port”; GET DATA(in-port) which returns a data value retrieved from the given “in-port” of the current plan; and SEND DATA (value, out-port) which sends the data value to the linked plan(s) on the “out-port”.

Creating a process plan block which produces a single value result can be done by following a pattern (see Figure 8). Firstly, it must have at least two parameters, “in-port” and “out-port”, which have default values “[plan name]:in” and “[plan name]:out”. Secondly, the plan block normally starts with the initialization of variables. Thirdly, a “repeat until” loop retrieves a sequence of values from the in-port. The loop condition is whether there is more data from the in-port using the scaffolding block “NO MORE DATA?”. Fourthly, inside the loop body, the first thing is to get a value from the in-port using the scaffolding block “GET DATA” and assign it to the variable initialized before the loop. Then it processes the value according to the algorithm of this plan, e.g. accumulating a running total for a Sum plan block. Finally, it sends the result to the out-port using the scaffolding block “SEND DATA”. The example of the Sum Plan block follows this pattern, and is built as in Figure 9.

Similarly, when a process plan block produces a sequence of values to its out-port and each of the output values is directly related to a value from the in-port, the process of building the plan is similar to that above, but the last step of sending a value to the out-port is inside the loop body. For example, the Even Number Plan block is created to produce even numbers from a sequence of values as in Figure 10.

Figure 7. Scaffolding blocks for constructing new plan block developed in BYOB

<table>
<thead>
<tr>
<th>Plan Name (in-port = [name of plan]:in), (out-port = [name of plan]:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize variable</td>
</tr>
<tr>
<td>repeat until &lt; NO MORE DATA? (in-port) &gt;</td>
</tr>
<tr>
<td>set (variable) to ( GET DATA (in-port))</td>
</tr>
<tr>
<td>Process variable according to the algorithm</td>
</tr>
<tr>
<td>SEND DATA (variable, out-port)</td>
</tr>
</tbody>
</table>

Figure 8. A pattern for a process plan

Figure 9. The sum plan details

Figure 10. The even plan detail

6 Evaluation

The approach described so far has been evaluated by having students use it. This section describes the evaluation setting, data collected, analysis method, and results.
6.1 Experimental Setting

The first author taught novice programming at a polytechnic (equivalent to TAFE in Australia) from 1997 to 2009. This teaching used a traditional approach where the constructs of a programming language (C++ before 2000, and Visual Basic from 2000) were introduced and then flowchart and pseudocode were taught as techniques for program design. Desk-checking, testing and debugging were also taught. Although the author had introduced various teaching innovations over the years (Hu 2004), these pre-date 2006, and during the period which we consider (2006-2009), there were no significant changes to the teaching content or method.

In 2011 the same author taught this course again using the proposed approach described in this paper. The first part of the course was still traditionally taught in terms of syntax, pseudocode, flowchart, desk-check, testing and debugging, but using BYOB. However, the second part followed the ideas proposed in this paper: teaching the ideas of goals and plans, and the process presented in Sections 4 and 5. The two parts were separated by a mid-term examination. After the mid-term examination the experimental method was introduced into the curriculum for four weeks, three hours per week, using the framework of the plan library and scaffolding blocks in BYOB.

In both 2011 and in previous years students were heterogeneous with different age groups and academic background. They were taught interactively in a small class (held in a lab class) with a mixture of theory and practice. The assessments included converting from flowchart to pseudocode and also from pseudocode to flowchart as well as programming. Similar programming questions were used from year to year, such as calculating the sum and (positive or negative) count, or the average of a sequence numbers.

In order to assess programming ability we collected the answers to the programming question in the final examination (and, for 2011, the mid-term test as well). We only considered the programming question (the examination also had other questions that did not assess the students’ ability to develop a complete program, the students could have done poorly on the programming question, but still may have done well on the examination overall). In all years (2006-2009, and 2011 in the mid-term and final examinations) the programming question was done on a computer, using a programming environment, rather than on paper. Note that a standard practice in polytechnics is that students who fail the examination are given a chance to re-sit the examination (some conditions apply). Where students took this opportunity, we only took results from their first attempt.

The students’ answers on the programming question were re-marked using a common marking rubric. The criteria used were:

1. identifying all the variables correctly, for example when calculating an average, important variables included “sum” and “count”;
2. correctly using fragments of key code, for example having code to count the number of values entered (but for this criteria we assessed the presence of essential code fragments, without requiring them to be combined correctly);
3. combining code fragments correctly; and
4. the final program being tested and bug free.

Note that these criteria are cumulative in the sense that having a tested and bug free final program required the presence of correctly combined code fragments. A final score out of 100 was calculated by summing these four criteria. The weighting used was 10 for the first criteria, 40 for the second, 30 for the third, and 20 for the final criteria. The distribution of students and their scores on the re-marked programming question in the examination is shown in both Table 2 and Figure 11. The wide range of performance, including both very low and very high scores, is typical of this sort of paper. The score results also support Caspersen and Kölling’s argument (2009) of ending up with “two groups of students” with or without their own process. The median and average (Table 2) are therefore not particularly useful, and the actual scores for the re-marked programming question (Figure 11) give a better picture of the performance of students in each year.

<table>
<thead>
<tr>
<th>Method</th>
<th>Year</th>
<th>Number of Students</th>
<th>Average Scores</th>
<th>Median value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>2006</td>
<td>13</td>
<td>33.3</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2007</td>
<td>16</td>
<td>53.8</td>
<td>82.5</td>
</tr>
<tr>
<td></td>
<td>2008</td>
<td>13</td>
<td>36.8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2009</td>
<td>8</td>
<td>39.4</td>
<td>22.5</td>
</tr>
<tr>
<td>Mid-term</td>
<td>2011</td>
<td>7</td>
<td>52.4</td>
<td>40</td>
</tr>
<tr>
<td>Experiment</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>2011</td>
<td>8</td>
<td>84.8</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2: Summary of Results

We note that there appears to be a ceiling effect: for the 2011 experimental method a number of students scored 100%. This is not the case for the other cohorts, who were assessed using an equivalent instrument. The impact of this ceiling effect is therefore that it reduces the difference between the experimental group and the other groups. In other words, if we had used a measurement instrument that did not exhibit this ceiling effect, we would expect to see a more significant difference between the 2011 experimental cohort and the other cohorts.

6.2 Statistics Methodology

The Kruskal-Wallis one-way analysis of variance by ranks (Kruskal and Wallis 1952) is a statistical test for measuring the likelihood that a set of samples all come from populations with the same probability distribution. It is a non-parametric test, which means that it does not make any assumptions about the shape of the underlying probability distribution. This is important for our study because the performance of novice programmers is known to not follow a normal distribution, so parametric analysis of variance techniques are not valid (especially as our sample sizes are small). The Kruskal-Wallis test is an “omnibus” test, which means that a single test is used to compare a number of statistics (the medians of the samples in this case). It is common to follow a significant omnibus test (i.e. if the null hypothesis is rejected) by a family of pairwise tests to gain more precise information about the
Hypotheses and Results

Our analysis aimed to determine to what extent the new approach for teaching programming made a difference. However, in order to draw conclusions about any difference between the 2011 results and results from earlier years being due to the new teaching approach we need to rule out alternative explanations.

In order to do this we developed two additional hypotheses. Firstly, we test the hypothesis that there is no significant variance within the years 2006-2009. If this hypothesis holds, then it suggests that differences due to variances in the cohort across years, or (slightly) different assessment questions are not significant. Secondly, we hypothesise that there is no significant difference between the earlier years and the 2011 mid-term test. If true, this suggests that any difference between the 2011 final results and earlier years are not due to differences between the 2011 cohort and the cohorts in earlier years. It would also suggest that the use of BYOB was not, of itself, sufficient to explain any difference. This finding would support Lister (2011), who argued that using Scratch and Alice will allow novice programmers to make better initial progress compared to using other languages, but that without a "pedagogical rethink of what should happen after these tools", there would still be issues when students are required to perform tasks that require transitive inference, such as realising that checking whether an array is sorted is equivalent to checking whether each pair of consecutive items are in order. In other words, visual programming languages may support novices to start programming. However, there must be a well defined pedagogical method to help novices bridge the gap to becoming experts. Finally, we hypothesise that there is a significant difference between the final results in 2011 and the results in previous years.

Hypothesis 1: The null hypothesis is that the examination scores from 2006 to 2009 come from populations with identical "locations". In other words, the median scores from each year are expected to not be significantly different.

The p-value of the examination scores from 2006 to 2009 by Kruskal-Wallis Test is 0.689 (> 0.05). It means there is not enough evidence to reject the null hypothesis. In other words, there are no significant differences in median scores in the past (2006-2009). This suggests that the examination results are similar in past years when teaching by the conventional method and that any differences of student cohort and variations of examination questions are not significant.

Hypothesis 2: The null hypothesis is that the mid-term examination scores of year 2011 and the examination scores from 2006 to 2009 also come from populations with identical "locations". In other words, the median scores in the past years and year 2011 mid-term are expected to not be significantly different.

The p-value from the student scores in the mid-term examination of year 2011 and those from 2006 to 2009 is 0.603 (> 0.05). This suggests that the examination results are similar in each year when still teaching by the conventional method despite differences of student cohort, variances of examination questions, and inconsistencies of computer languages (VB vs. BYOB).

Hypothesis 3: The null hypothesis is that the final examination scores from year 2006 to 2009 and year 2011...
come from populations with identical “locations”. In other words, the median scores of year 2011 based on the experimental method and years in the past based on the conventional method are hypothesised to not be significantly different.

The p-value from the student scores in the past examinations (2006 - 2009) and those from the final examination in 2011 is 0.031 (< 0.05). This shows a significant difference between the median examination score in 2011 and the examination scores in previous years, and provides evidence for rejecting Hypothesis 3. Since there is no significant difference in the past (Hypothesis 1), it implies that the difference comes from year 2011. Furthermore, since there is no significant difference between the past and the 2011 mid-term examinations (Hypothesis 2), it suggests that the difference comes from changing to different teaching methods after the 2011 mid-term examination.

Since the above tests did not clearly indicate which year(s) causes the difference, we conducted paired comparisons. Because the Kruskal-Wallis Test on the data from all years except 2011 showed no significant difference (Hypothesis 1) we only considered paired comparisons between 2011 and other years (2006 -2009). We used Holm’s sequential Bonferroni method (Holm, 1979) in order to reduce the chance of any type 1 errors. With this method the p-value (from smallest to largest) by Mann-Whitney U Test for each paired comparison needs to be smaller than its threshold p-value to be significant, where the threshold p-value is divided by (C-i+1), making the test more conservative. For example, the first test, which is the one with the lowest U-test p-value (comparing 2011 and 2006) has a threshold p-value of 0.05/4; the second comparison has a threshold of 0.05/3, etc. The results of these Mann-Whitney tests (see Table 3) indicated significant differences of examination scores between the year 2011 and each individual year from 2006 to 2009.

<table>
<thead>
<tr>
<th>Paired Comparisons (C=4)</th>
<th>Threshold p-value 0.05/(C−1 + 1)</th>
<th>p-value by U test</th>
</tr>
</thead>
<tbody>
<tr>
<td>between 2011 to 2006</td>
<td>0.013</td>
<td>0.003 (&lt;0.013)</td>
</tr>
<tr>
<td>between 2011 to 2008</td>
<td>0.017</td>
<td>0.01 (&lt;0.017)</td>
</tr>
<tr>
<td>between 2011 to 2007</td>
<td>0.025</td>
<td>0.021 (&lt;0.025)</td>
</tr>
<tr>
<td>between 2011 to 2009</td>
<td>0.05</td>
<td>0.025 (&lt;0.05)</td>
</tr>
</tbody>
</table>

Table 3: Comparing Paired Examination Scores

To summarise, through the analysis of past examination results we have seen a statistically significant improvement in student performance using our new approach. We have provided evidence that the difference is not due to variation in the cohort, in the examination questions, or due to the use of BYOB.

7 Conclusion

Our research suggests that the experimental teaching method which combines using goals and plans, a well-defined process, and a visual notation, has the potential to significantly improve learning of programming skills. A key feature of our approach is that we provide a detailed process that guides novices through the process of developing a program, using goals and plans. Another key feature that distinguishes our approach from other work is that our representation for goals and plans is integrated into a (visual) programming language, and that this integration is done in such a way as to allow an intermediate program to be executable. Once plans are defined and linked, the resulting program can be executed, even though the plan bodies have not yet been merged to produce a final program. We see this as a significant advantage for a number of reasons. Firstly, it provides earlier feedback, and supports testing and debugging. Secondly, plan merging is known to be difficult (Soloway 1986), and by allowing novices to obtain feedback on their unmerged designs, they can improve their design without having to perform plan merging. Finally, it allows novices to distinguish between errors in their design and errors introduced by faulty plan merging.

A limitation of the evaluation is that we considered the new approach as a package. While this makes sense in that it is the combination of factors that makes the approach effective, it is possible that some factors are less important than others. For example, students in 2011 had a plan library provided. This clearly assists with completing a programming task, and we cannot say to what extent the improvements in student performance were due to this factor. Another limitation of our evaluation is that we only used a programming question in an examination to measure performance. We argue that an examination is an appropriate choice because it is conservative: it tends to underestimate ability (due to time constraints), and it eliminates the possibility that exists in assignments that students obtained significant assistance from peers, family, friends, or tutors. We also did consider assignment results, and found that there were no significant differences between 2011 and earlier years. Another limitation of the evaluation is that we had only a limited number of students in each year. Future work could include evaluating this framework with more students. Other potential issues with the evaluation are that the course was taught by an author of this paper, who might be expected to be enthusiastic about the new approach. We argue that while this is certainly true, the author was equally enthusiastic about their past teaching and that when conducting teaching in 2006-2009, the approach described in this paper had not yet been developed, or even conceived. Finally, the students in 2011 were aware that they were being taught using a modified experimental method (since they had to sign ethics approval forms), but any form of Hawthorne effect would be expected to apply to the whole course, including performance in the mid-term test, since students did not know which part of the course was traditional and which was novel.

There are a number of directions for future work. At the moment we have defined visual notation for both goals and plans, but only the visual notation for plans has been integrated into Scratch. One area for future work is therefore completing the integration of the representation of goals into Scratch. Additionally, although the plan merging process is well-defined, it is somewhat complex, and one key area for future work is therefore how to support novices in performing plan merging. Because we want novices to eventually move away from using our framework, it is important that this support not be in the form of a “wizard” that does the merging of plans without the student gaining any insight into the merging process.
8 References


