A Process for Novice Programming Using Goals and Plans

Minjie Hu
Department of Information Science
University of Otago
PO Box 56, Dunedin 9054
minjiehu@infoscience.otago.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
scranefield@infoscience.otago.ac.nz

Abstract
We propose to improve the teaching of programming to novices by using a clearly-defined and detailed process that makes use of goals and plans, and a visual programming language. We present a simple notation for designing programs in terms of data flow networks of goals and plans, and define a detailed process that uses this notation, and that ultimately results in a program in a visual programming language (BYOB). Results from an evaluation are presented that show the effectiveness of this approach.

Keywords: Goal, Plan, Process of Programming.

1 Introduction
A range of studies across institutions and countries have observed that novices struggle in introductory programming (Lister et al. 2004, McCracken et al. 2001). Accordingly a wide range of approaches have been proposed to improve novices’ learning of programming. For example, the problem-solving approach emphasises the development of problem-solving skills connected to programming (Pears et al. 2007); problem-based learning (PBL) is based on solving a “large real-world” problem collaboratively in groups (Kay et al. 2000); collaborative learning provides support and enhances communication in learning environments to promote students’ high level cognitive skills (Rößling et al. 2008); psychological analysis considers the mental models of novices, and proposes various concrete conceptual models to help novices to understand programming (Mayer 1981, Winslow 1996); programming visualisation provides visual support towards the development of viable mental models and engages novices in an active learning activity to improve their understanding of programming and help their learning (Ben-Ari 2001, Naps et al. 2003); and game programming attracts, motivates and engages novices to learn programming by using computer games as a subject based on multimedia, pre-developed libraries or micro-worlds (Guzdial and Soloway 2002, Kölling and Henriksen 2005).

However, these approaches have not been widely adopted. Although some approaches (e.g. PBL) have been demonstrated to be highly effective (Kay et al. 2000), they are quite costly to introduce. By contrast, we aim to develop an approach for teaching novice programming that is both effective and cheap to introduce. Specifically, we propose to combine three aspects: the use of a visual programming language; the use of goals and plans; and the employment of a clear well-defined process with feedback.

Recently, visual programming languages (VPLs), such as Scratch and Alice, have been used to teach novice programming. Programs are built by dragging and dropping statement blocks, which helps to prevent syntax errors and enables students to make better early progress (Lister 2011). The philosophy of using VPLs to teach novices is to “let them play first, let them achieve something, … and then sneak the explanations in” (Utting et al. 2010, p7). However, there are concerns that students might “simply mess around and never focus towards any goal” (Utting et al. 2010, p4). In other words, students may learn to program by trial and error, rather than by following a systematic approach. Therefore, they need guidance, that is, a process, for how to program.

In order to give guidance to novice programmers, we take as our point of departure the work on goals and plans. A goal is a certain objective that a program must achieve in order to solve a problem (Letovsky and Soloway 1986), and a plan (Spohrer, Soloway, and Pope 1985) corresponds to a fragment of code that performs actions to achieve a goal. In the 1980s, Soloway (1986) and his colleagues (Letovsky and Soloway 1986, Spohrer, Soloway, and Pope 1985) discovered that experts have strategies to solve problems using their own libraries of plans. They advocated structuring these libraries in terms of goals and plans, and teaching strategies for using these libraries. Subsequently, educators have been attempting to introduce goals and plans as a means of structuring the development of programs (de Raadt 2008, de Raadt et al 2006, Guzdial et al. 1998, Soloway 1986). For example, various template-based approaches for using program fragments were proposed such as “pedagogical programming patterns” (Porter and Calder 2003) and “programming strategies” (de Raadt 2008). However, this body of work did not provide a detailed process for using goals and plans in program development.

For the reasons outlined above, there is therefore an opportunity and need to provide a detailed step-by-step
process for programming by novices (Caspersen and Kölling 2009). There have been a number of approaches that focus on teaching novices a process for programming. Passis (1990) proposed that the programming process be broken down into a series of well-defined steps, and that it is important to provide feedback from each step. Providing feedback at each step was considered to be critical in giving students confidence. In fact, feedback is at the heart of test-driven development, and Janzen and Saiedian (2006) recently proposed to improve teaching by using “test-driven learning”. However, none of these papers provided a detailed process that could be taught to novices.

A number of detailed processes for teaching novices have been proposed. For example, Programming by Numbers (Glaser, Hartel, and Garratt 2000) and TeachScheme (Felleisen et al. 2004) both provide a clear process for creating the smallest components of functions, using stepwise refinement. Both approaches are data-driven and more suited to functional programming languages than to mainstream procedural languages. Another process that has been proposed is STREAM (Caspersen and Kölling 2009), which aims to teach novices a process for object-oriented programming. However, none of these proposed processes included the use of goals and plans (or of a VPL).

Our research therefore focuses on the development of a programming process for novice programmers using goals and plans in a visual programming environment. Our programming process aims to be clearly defined, detailed, iterative, incremental, and to provide feedback at every step. Our previous work (Anonymous 2012) presented an overview of our approach, whereas this paper provides a more detailed presentation of the process.

2 Design Notation

Before providing a process for program development using goals and plans, we first need to define a design notation, which is used to capture the results of the first step of the process.

Soloway and colleagues proposed that novices develop a structure for their program in terms of a tree of goals. This tree is progressively refined, and ultimately, goals are realized as combinations of plans. Initially, Soloway (1986) proposed three ways of combining plans: sequential (Plan B begins after Plan A finishes), nested (Plan B is used as one of the steps within Plan A), and interleaved (the steps of Plan A and B are merged with each other). Subsequently Ebhrahimi (1992) proposed an additional plan combination: branching (Plan A uses either Plan B or Plan C, depending on a condition).

The term “interleaved” plans above does not indicate plans that have been interleaved, but rather plans that will need to have their implementations merged to form an executable single-threaded program. For example, consider computing an average of a sequence of numbers, which has been designed using a plan for computing the sum of the sequence, and another plan for computing the count of the sequence (number of items). An executable (procedural) program needs to read each input, and then process it, including updating both the sum and the count, before the next input value is read. For example, see the right part of Figure 7, which shows an executable procedural program (in BYOB1) where three plans (Input, Sum, and Count) are interleaved.

However, by adopting a data flow based representation, we can avoid the need for merging before any execution can be done. We model plans as consuming and producing sequences of data. For the above example, if we have a data flow buffer between plans, then we can execute the Input plan, store the results in a buffer, execute the Sum plan to completion, then execute the Count plan to completion, and finally proceed to compute the average by dividing the sum by the count. In other words, by using a data flow model, we can execute unmerged plans. This is a significant difference between control flow and data flow models when using goals and plans, because it enables novices to receive feedback before plans are merged into a final (procedural) program.

We therefore propose a notation where goals and plans are represented by icons, and are linked by arrows (denoting data flows between them). Goals can be categorized into three types: input, output, or processing (Figure 1). A simple program might have only one goal of each type and achieve these goals in sequence, but more typically a program would have multiple processing goals.

Figure 1: Example of three basic goals

A data flow is a sequence of values that “flow” between goals. Each data flow is represented by an arrow that links two goals. In the typical case, a data flow has a single source and single destination, and links two goals. It is also possible for a data flow to have two destinations, indicated with a “fork” notation (e.g. Figure 4). We make the assumption that each goal has a single out port and we use arrows to show the direction of the flow. This means that we do not need to have named ports on goal diagrams, which simplifies the initial design stage for students.

The second design stage is to produce a plan diagram corresponding to the goal diagram. Since a plan is a code segment that accomplishes a programming goal, it is visualised as a box with double lines on both sides like a sub-program icon in flowchart notation. Plan icons are used to replace all goal icons in the goal diagram, yielding a plan diagram (Figure 2). At this stage, plan ports are added and named. A data flow can be accessed from within plans by two ports: from a source (“out”) port of one plan to a destination (“in”) port of another plan. Ports are identified and named by the combination of the name of the plan and the function of the port. For example, the port name Input:out represents the out port of an Input Plan.

---

1 Build Your Own Blocks (BYOB) is a variant of Scratch (see http://scratch.mit.edu). BYOB permits users to build new “blocks”. Each new block can include a procedure. See http://byob.berkeley.edu6
We distinguish between ports that are associated with a data flow that only involves a single value, and those that are the end points of a data flow with a sequence of values. For example, the first data flow in Figure 2 links the port Input:out to the port Sum:in and can contain a sequence of values. However, the data flow from Sum:out to Output:in will only contain a single value.

### 3 Programming Process

The process of programming that we propose consists of five steps: (1) analysing goals and plans; (2) mapping the plan network to BYOB using plan blocks (where each plan is mapped to a BYOB plan block); (3) expanding plan blocks; (4) merging the expanded plan details; and (5) simplifying the merged details (Figure 3). Each step includes three or four sub-steps.

In order to provide early and frequent feedback, we adopted the test-driven learning approach. Before tackling the first step, students have a “step zero” (not shown in Figure 3) in which they study the question, and specify test cases (both input and correct output). For example, if the problem is to compute the average of input numbers, then a student might define the following two test cases: given 1, 2, and 3 the average is 2; and given 2, 3, 7 and 8 it is 5.

After each step, the student checks that the results are correct, i.e. that the design produces the expected answers for the test cases. For the first step, this checking is done by a manual desk-check, but for the following four steps, all the test results come from executable programs.

The programming process that we describe below guides the student through an incremental development process that proceeds from goal and plan concepts, to plan block design, to intermediate program, and to final code. The process is illustrated using the following example, which was originally used by Soloway (1986) to analyse goals and plans:

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

### 3.1 Analysing Goals and Plans

Goal analysis starts by identifying what goals the program needs to achieve. Typically a program includes at least one input, one output, and some number of processing goals, some of which may need hierarchical refinement. For the above example, the first goal is to input the values. Following this, a “compute average” goal was initially required. However, the average goal needs the sum and count of the input. Hence, the “compute average” goal can be decomposed into three goals, sum, count, and divide, where both sum and count receive the same data flow from the input goal (in parallel), and send their results to the divide goal (also in parallel). Finally, the result of the divide goal is sent to
the last goal (to output the result). Therefore, five goals (1–5) of three types are identified and presented using visual notation in Figure 4. The relationships between these goals were also identified and the goals are linked by five data flows (A–E in Figure 4). Note that the numbers 1–5 and the letters A–E are only added here for ease of explanation, i.e. they are not part of the notation.

In general, the development of the goal diagram is an incremental refinement process, in which the processing goals are hierarchically decomposed. Once goals are refined to a level where they are sufficiently fine-grained, they can be mapped to plans in a one-to-one manner (i.e. each goal becomes a plan), resulting in a plan network diagram (Figure 5). For teaching novice programming, a sufficiently fine-grained decomposition of goals means that the decomposed goals correspond to BYOB plan blocks in a provided plan library or that they can be implemented simply. For the example in Figures 4 and 5, five plan icons have been used to replace the five goals.

The next step is to realise the data flows by defining ports. In simple cases, a port name can be identified by the combination of plan name and type of the port (“in” or “out”). For example, the port on the left2 of “Sum Plan” is named Sum:in; and the port on the right is named Sum:out. In cases where a plan has multiple incoming data flows, its graphical representation shows multiple in ports with different names. For example, the two in ports of the Dividing Plan are identified as dividing:in.dividend and dividing:in.divisor.

After mapping from the initial goal diagram to a plan network, a desk-check table is used to test whether or not the first step of analysis is correct (see Table 1). The table consists of two parts. The first part contains the first two columns of the table: Test Cases and Predicted Answers. The contents of these two columns are based on the test cases specified in step zero: the test case cells are the input of the test case, and the predicted answer is the expected output. The second part of the table comprises the rest of the columns, and is based on the plan network. Each column represents one port within the plan network. The cell under the “in” column of a plan is filled with a copy of the data from the relevant “out” column, i.e. the “out” port that is linked to it by a data flow. For example, the “in” column for the Sum Plan is simply a copy of the “out” column of the Input Plan, since Input’s out port is linked to the Sum’s in port. The cells under the “out” column of a plan are filled by computing the corresponding output of a plan, given its input, for example the “out” column for Sum Plan is the sum of its inputs. Hence, the second part of the table records the data flow through the plan network.

<table>
<thead>
<tr>
<th>Test Cases:</th>
<th>1, 2, 3, -1</th>
<th>2, 3, 7, 8, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Answers:</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Input</td>
<td>out</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>6</td>
</tr>
<tr>
<td>Count</td>
<td>in</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>3</td>
</tr>
<tr>
<td>Dividing</td>
<td>in.dividend</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>in.divisor</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>2</td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Example of a desk-check for data flow in plan ports

The first step of analysis is completed after the outputs from the last column are the same as the predicted answers in the second column for every test case. Otherwise, the analysis has to be corrected. The plan network produced is used in the next step, where it is mapped to BYOB, using plan blocks.

3.2 Encoding Plan Network Using BYOB Plan Blocks

Following the confirmation of the correctness of the goal and plan analysis by desk-checking, the diagram of the plan network (Figure 5) can be mapped to an executable plan network (Figure 6). The process for doing this is fairly straightforward and mechanical. Each plan icon is replaced by a plan block in BYOB, and every data flow is mapped to a “scaffolding block” (Link <<out port name>> to <<in port name>>) to link an out port to an in port. Note that the order of the plan blocks (1–5 in Figure...
Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia

6) does matter: the plan blocks need to follow the order of the arrows in the plan network (Figure 5). For example, in this case the Input plan must be first, followed by the Sum and Count plans (in either order), and then the Dividing plan and finally the Output plan.

![Figure 6: Example of an executable plan network in BYOB](image)

We map plan icons to plan blocks by considering the diagram of the plan network from the previous step. Using BYOB, each plan can be implemented as a plan block with arguments that are ports to receive and/or send data flow. Processing plans (i.e. plans other than an Input or Output plan), encapsulate a procedure to receive data flow from their in ports, to process the data flow, and then send the results to their out port.

Individual plan blocks are identified from our plan library developed in BYOB. If a plan block does not exist, then the student must build it based on similar library developed in BYOB. If a plan block does not exist, then send the results to their out port.

Dividing plan and finally the Output plan.

Individual plan blocks are identified from our plan library developed in BYOB. If a plan block does not exist, then the student must build it based on similar pattern of existing plan blocks in the library. For example, considering the plan network in Figure 5, plan blocks 1–3 and 5 can be found in the provided library, but plan block 4 (“Dividing Plan”) is not in the provided library. However, the library has a “Multiplying Plan” block, which is similar and can be used as a template for developing the “Dividing Plan” block.

In order to represent a plan network in BYOB we use a number of scaffolding blocks. Note that eventually all the scaffolding blocks will be removed from the final program. There are three data flow scaffolding blocks which are used to deal with data flow within a plan. They are named “NO MORE DATA <<port>>”, “GET DATA <<port>>”, and “SEND DATA <<datum>> <<port>>”. The first is used to find out if there is any datum in the input port of the current plan. The second is used to get a datum from the input port. The last (SEND DATA) is used to send a result of the current plan to its output port.

There are also linkage scaffolding blocks which are used to define the linkages between plan blocks. These are placed at the start, between “Begin Links” and “End Links” blocks. Each Link block specifies a linkage from an out port (source) to an in port (destination). If the current plan output port is linked by a scaffolding block

to an input port of another plan, this data flow will be sent to the linked plan through its input port. Each link in the plan network diagram (Figure 5) is directly mapped to a Link block in BYOB (Figure 6, indicated with letters A–E). For example, data flow “A” is mapped to the first two Link blocks. The first Link block links the out port of the Input Plan (Input:out) to the in port of the Sum plan (Sum:in). The second Link block links the same out port to the in port of the Count plan (Count:in).

Note that we provide default port names in each plan block that combine the plan name and port function, for example “Sum:in” and “Sum:out”. Therefore, students do not need to create port names, and can fill in the port names in the Link blocks by copying from the plan blocks. However, when students use same plan block more than once in their program, they have to change the default port name for different copies of the same plan. For example, if a second copy of the Sum Plan block is used, its default port names must be renamed from “Sum:in” and “Sum:out”. Correct and consistent use of port names is essential for correctness, and the internal implementation of the Link blocks tests for this, and gives a message if the port name filled in the Link block does not match the spelling of its original name in the plan block.

The result of this process (Figure 6) is fully executable, and can be tested and debugged. Note that the execution makes use of buffers: in this example (see the left side of Figure 7), the Input plan runs to completion, collecting all the inputs, then the Sum plan runs to completion (reading from a behind-the-scenes buffer) and computes the Sum of the input, followed by the Count plan counting the number of input values, and so on. This is quite different to how a final (procedural) program executes: a single input is read, and a running sum and count updated before dealing with the next input value.

The testing of this step is part of the evaluation and testing schedule for the whole programming process. As students proceed through the process in Figure 3 they maintain a checklist (Table 2). The first three columns are filled according to the results in Table 1. The remaining columns correspond to the steps in the process. After each step, the results from testing are recorded and compared with results from the earlier steps in the process. In this step, the testing and debugging results are filled in the fourth column (Step 2), and Step 2 will be considered to be completed if the test results in the Step 2 column are the same as those in column Step 1.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>1, 2, 3, -1</th>
<th>2, 3, 7, 8, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Answers</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Results from analysis of goals and plans (Step 1)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Results from mapped plan blocks (Step 2)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Results from expanded plan details (Step 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Results from merged details (Step 4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Results from final program (Step 5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Example of the test schedule
### 3.3 Expanding Plan Blocks

Expanding plan blocks means replacing each plan block with the defined details within it (see the left side of Figure 7). Since data flow blocks (“GET DATA”, “SEND DATA”, and “NO MORE DATA?”) inside every plan block contain local parameters “in-port” and “out-port” to refer to ports of their hosting plan block, after expanding from a plan block, these parameters must be changed to the plan port names in order to work with the linkage using plan ports. For example, the blocks details within the Sum plan (which are not shown in Figure 7) include the block “repeat until NO MORE DATA? in-port”, where “in-port” is the first parameter of the plan block. When replacing the Sum plan block with its plan details we replace “in-port” with the value of the plan block’s first parameter, namely “Sum:in” (Figure 6) yielding the statement “repeat until NO MORE DATA? Sum:in” (see the numbers part 2 in the left side of Figure 7).

To help explain this process we have provided students with video clips of a screen capture that demonstrates how to firstly duplicate plan details from each plan block and then how to replace the parameters by copying-and-pasting a port name from the plan block. Whereas the previous two steps require human thought and creativity, this step is purely mechanical and could be automated in future work. Note that the expanded program is also executable and testable.

### 3.4 Merging Expanded Plan Details

Merging expanded plan details aims to combine the details from different plans into one program in which data flow (and the associated use of buffers) is eliminated, and in its place, a single datum is sent and received between plans. In other words, traditional variables, rather than buffers, are used to communicate data between plans. Common variables are also shared between plans (see the middle part of Figure 7). Since plans all follow the same pattern (iterating, reading items from their input port, and dealing with items one at a time), they can be merged by following three steps, which are demonstrated to students with video clips of examples. The first step of merging plan details is to collect all the blocks that initialise variables by setting or inputting initial values, and put them immediately after the “End Links” block, i.e. at the start of the merged program. For example, in the middle of Figure 7, the first two statements, “set Sum to 0” and “set Count to 0”, as well as the 3rd and 4th statements, “ask” and “set” to input the initial value for variable Number, are placed immediately after “End Links”.

The second step is to combine loops that share the same data flow. In this situation, the first loop is used to generate the data flow, while the other loops receive this data flow. Hence, the bodies of the other loops can be moved within the first loop. Specifically, consider the case where output port portA is linked to input port portB, and we have the following two loops:

```
Repeat until <condition>
  <body of first loop part 1>
  SEND DATA (value, portA)
  <body of first loop part 2>
End repeat
Repeat until NO MORE DATA(portB)?
  Set Var to GET DATA (portB)
  <body of second loop>
End repeat
```

Then both loops will execute the same number of times because the second loop executes once for each data item sent in the first loop. Therefore, the second loop can be eliminated by moving its body inside the first loop:

```
Repeat until <condition>
  <body of first loop part 1>
  SEND DATA (value, portA)
  <body of first loop part 2>
End repeat
```

This assumes that there are no common variables between the loop bodies, which can be ensured by renaming. Since the loop bodies originate in different scopes, there cannot be common variables. However, there may be variables which use the same name (in different scope), and bringing them into the same scope would require renaming to avoid the distinct variables being conflated.
For example, since the loop of the Input Plan generates a data flow to both the Sum Plan and the Count Plan, three loops (see parts 1, 2 and 3 in the left side of Figure 7) are merged under the loop condition from the Input Plan loop (Repeat Until Number = -1). The loop bodies from both Sum Plan and Count Plan are put after “SEND DATA Number Input:out” and before the blocks for inputting the next value of Number (see the numbered parts (1–2–3–1) of the program in the middle of Figure 7). Note that blocks outside of each loop body, such as “SEND DATA Sum Sum:out” and “SEND DATA Count Count:out”, are still kept outside of the merged loop (see the final two blocks in the middle of Figure 7).

The third step is to remove the loop control where there will only be a single value in a data flow. For example, the loop controls from the Dividing Plan and Output Plan can be removed, since the input data flows to these plans only have single values (as shown in Figure 5, and confirmed in Table 1). Once more, the merged plan details are executable and testable. Table 2 is used to check whether the testing results from Step 4 are the same as those from previous steps.

3.5 Simplifying the Merged Details

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

When a variable has its value sent to an output port, and subsequently another variable receives the same value from a linked input port, the second variable should be consistently renamed to match the first one. When we have code of the form:

\[
\text{LINK } p_1 \text{ } p_2 \\
\ldots \\
\text{SEND DATA } (v_1, p_1) \\
v_2 := \text{GET DATA}(p_2) \\
<\text{code referring to } v_2>
\]

Then the variable \(v_2\) receives its value (via the SEND and GET) from \(v_1\), and can be renamed to \(v_1\):

\[
\text{LINK } p_1 \text{ } p_2 \\
\ldots \\
\text{SEND DATA } (v_1, p_1) \\
v_1 := \text{GET DATA}(p_2) \\
<\text{code referring to } v_1>
\]

For example, consider “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 renamed to Sum. Similarly, variable Number2 is replaced by variable Count.

This renaming of variables means that the SEND and GET blocks become redundant and can be removed, leaving only variables and control flow blocks, which are independent from scaffolding and data flow blocks. Therefore, the last step is to remove all the scaffolding blocks, both those used to define links (“Begin Links”, “Link”, and “End Links”), and those used to specify data flow (“NO MORE DATA?”, “GET DATA” and “SEND DATA”). This results in the final program shown on the right of Figure 7. At the end of this process, the final program is tested and the last column of Table 2 shows the test results in Step 5, which should be the same as those in the previous column.

4 Evaluation

We evaluated our approach by comparing the answers to a programming question from exams in an introductory programming course at Tairawhiti Campus, Eastern Institute of Technology in New Zealand. We collected answers from the final exams in the course from 2006 to 2009, and for 2011 collected answers from the final exam and the mid-term test (Hu, Winkoff, and Cranefield 2012). Note that the programming questions used in the exam were similar across years, for example, calculating the sum and (positive or negative) count, or the average of a sequence numbers, and are thus comparable. Also, note that the programming questions in this course’s exams are done on a computer, rather than on paper.

In all years the course was taught by the first author of this paper. In the institutes of technology and polytechnics in NZ, the introductory programming course is delivered as a total of nine week module for the first year diploma programme. Each week had a three hour mixture of teaching and exercises in a computer lab. The course outlines are listed in Table 3. From 2000–2009 the course taught programming using Visual Basic (VB) (and a conventional approach). In 2011 the course used BYOB. The 2011 course retained a conventional approach for the first half of the course, but adopted our proposed process and tool for the second half (see Table 3).

All the answers on the programming question were remarked using the same criteria: identifying variables, using fragments of key code, combining fragments, and being bug free. The summaries of exam results are shown in Table 4. In order to establish a causation (i.e. students did better because of the new method) we consider a range of possible alternative explanations for the performance improvement, and rule them out. We compare data from students using the new method with data from students using the old method, and given an observed difference, we rule out other possible causes, such as different student cohorts from year to year, different exam questions, or changes of computer languages (VB to BYOB). We do this by considering a number of hypotheses.

Our first hypothesis is that the scores of the programming question in final exams for the conventional approach (2006–2009) do not show significant differences (more precisely: come from populations with the same probability distribution). This is the case (\(p = 0.689 > 0.05\), see Table 5) and so we conclude that changes in cohort from year to year, and in exam questions from year to year do not make a significant difference.

Our second hypothesis is that the scores for the conventional approach (including both 2006–2009, and
also the 2011 mid-term exam\(^6\) do not show significant difference. Again, this is the case \((p = 0.603 > 0.05)\), which suggests that the 2011 cohort is not significantly different to earlier cohorts, and also that the use of BYOB rather than VB is not in itself the cause of a significant change in performance in the exam.

![Table 3: Course outline by years](image)

<table>
<thead>
<tr>
<th>Wk</th>
<th>2006 – 2009</th>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduc-</td>
<td>Introduc-</td>
</tr>
<tr>
<td></td>
<td>tion to</td>
<td>tion to</td>
</tr>
<tr>
<td></td>
<td>Visual</td>
<td>BYOB</td>
</tr>
<tr>
<td>2</td>
<td>Input,</td>
<td>Making</td>
</tr>
<tr>
<td></td>
<td>Process</td>
<td>Decision</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>Flowchart</td>
</tr>
<tr>
<td></td>
<td>Output</td>
<td>Flowchart</td>
</tr>
<tr>
<td>3</td>
<td>Making</td>
<td>Nesting</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>More</td>
<td>Selection</td>
</tr>
<tr>
<td></td>
<td>about</td>
<td>Flowchart</td>
</tr>
<tr>
<td></td>
<td>Making</td>
<td>Desk Check</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>BYOB</td>
</tr>
<tr>
<td>4</td>
<td>More</td>
<td>Review</td>
</tr>
<tr>
<td></td>
<td>about</td>
<td>Nesting</td>
</tr>
<tr>
<td></td>
<td>Making</td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>Decision</td>
<td>Repetition</td>
</tr>
<tr>
<td>5</td>
<td>Repeating</td>
<td>Analysis</td>
</tr>
<tr>
<td></td>
<td>Actions</td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
<td>Problems</td>
</tr>
<tr>
<td>6</td>
<td>Integration</td>
<td>Steps</td>
</tr>
<tr>
<td></td>
<td>of</td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>Selectio-</td>
<td>Solving</td>
</tr>
<tr>
<td></td>
<td>n and</td>
<td>Problems</td>
</tr>
<tr>
<td></td>
<td>Repetiti-</td>
<td>by</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>Provided Plans</td>
</tr>
<tr>
<td>7</td>
<td>More</td>
<td>Build Your</td>
</tr>
<tr>
<td></td>
<td>about</td>
<td>Own Plans</td>
</tr>
<tr>
<td></td>
<td>Integration</td>
<td>Solving</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Problems from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Goal to</td>
</tr>
<tr>
<td>8</td>
<td>Revision</td>
<td>Revision</td>
</tr>
<tr>
<td>9</td>
<td>Test</td>
<td>Written and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>practical</td>
</tr>
</tbody>
</table>

Table 3: Course outline by years

Our last hypothesis is that the new method does make a difference, i.e. that including students’ performance in the final exam in 2011 (i.e. after being taught our new approach) will result in a significant difference. This is the case \((p = 0.031 < 0.05)\), and since we have excluded a change in cohort, or programming language, or exam question, we conclude that our method has made a significant difference.

![Table 4: Summary of student results](image)

<table>
<thead>
<tr>
<th>Student Groups</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final exam from 2006 to 2009</td>
<td>0.689 ( &gt; 0.05)</td>
</tr>
<tr>
<td>Final exam from 2006 to 2009 and mid-term exam 2011</td>
<td>0.603 ( &gt; 0.05)</td>
</tr>
<tr>
<td>Final exam from 2006 to 2009 and final exam 2011</td>
<td>0.031 ( &lt; 0.05)</td>
</tr>
</tbody>
</table>

Table 4: Summary of student results

Having determined that including the 2011 final exam leads to a significant difference (i.e. rejecting the null hypothesis), we would like to find out which medians of examination scores are different. We performed a family of pairwise comparisons using the Mann-Whitney U test and Holm’s sequential Bonferroni adjustment to reduce the chance of any type 1 errors. We only consider the four comparisons between the samples from 2011 and each of the earlier years (see Table 6). This is because 2011 is the year in which the intervention we wish to measure was applied. We made three hypotheses to the sample data summarized as follows. The Mann-Whitney U Test result of each paired comparison to 2011 is smaller than its threshold \(p\)-value, which indicates significant differences of examination scores between the year 2011 and each individual year from 2006 to 2009. Therefore, the evaluation has shown a statistically significant improvement in student performance using our new approach. The difference is not due to variation in the cohort, in the examination questions, or in the use of BYOB.

![Table 5: Kruskal-Wallis H test\(^6\) results](image)

<table>
<thead>
<tr>
<th>Paired Comparisons</th>
<th>p-value (Holm-Bonferroni threshold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final exam between 2011 to 2006</td>
<td>0.003 ( &lt; 0.013)</td>
</tr>
<tr>
<td>Final exam between 2011 to 2008</td>
<td>0.01 ( &lt; 0.017)</td>
</tr>
<tr>
<td>Final exam between 2011 to 2009</td>
<td>0.021 ( &lt; 0.025)</td>
</tr>
<tr>
<td>Final exam between 2011 to 2009</td>
<td>0.025 ( &lt; 0.05)</td>
</tr>
</tbody>
</table>

Table 5: Kruskal-Wallis H test\(^6\) results

There were two limitations in the evaluation. However, even though the number of students was low, there was still a clear and statistically significant result. There was also a ceiling effect (where for the 2011 cohort

\(^6\) Since we had a small group of students in each year (between 8 to 16) and the performance of novice programmers is known to not follow a normal distribution, we used a non-parametric statistical analysis of variance technique, which does not make any assumptions about the shape of the underlying probability distribution. The Kruskal-Wallis one-way analysis of variance by ranks (H test) is a statistical test for measuring the likelihood that a set of samples all come from populations with the same probability distribution.
five out of eight students produced programs that were awarded full marks). However, this ceiling effect actually reduces the difference between the experimental and conventional groups, and there would be a more significant improvement if we used an instrument that did not have a ceiling effect. Overall, the evaluation results are significant, but further evaluation would help to strengthen the results.

5 Conclusion

We have introduced a well-defined, iterative and incremental program development process for teaching novice programming. The process provides a guideline for novices to develop from the concepts of goals and plans to final code in a visual environment. The process includes five major steps (Figure 3), which guide the student through a process of stepwise refinement. Each step has strategies, and heuristics to guide novices. A significant difference with existing process approaches is that our process includes feedback from every step rather than having one round of feedback from the final program. This regular feedback from every intermediate step encourages students to continue to progress “from victory to victory” in the next step. Our research suggests that the experimental teaching method proposed, with a well-defined process, use of goals and plans, and a visual notation, has the potential to significantly improve learning of programming skills.

Note that we are aiming to teach generic programming in a way that leads to further computer courses. This is why we focus on problems that are more representative of the sort of algorithmic programming done in later courses, rather than the sort of applications that BYOB is typically used for. In other words, BYOB is merely a vehicle, and the process is applicable to other programming languages. One area for future work is to assess whether the cohort that did the course in 2011 (with the new teaching method) did better, worse, or the same in subsequent programming courses. We have done a preliminary comparison of the overall exam mark in the second programming course (PP590) for the 2006-2011 cohorts. However, although the PP590 exam marks for the 2011 (experimental) cohort are higher (average of 53.857 for 2011, compared with 34.1 across 2006-2009 [2007 is lowest with 23.67, 2008 highest with 44.308]), the difference is not statistically significant. Note that the PP590 exam typically includes both questions that involve programming, and questions that assess knowledge rather than programming skill (e.g. "what is pseudocode?"). This means that the overall exam mark is not a good measure of programming ability. Unfortunately, we do not have the marks for individual questions, only the total exam mark.

As noted earlier, where the library is missing a required plan block, the student must develop it themselves by modifying similar blocks (e.g. multiplication to division), or by using a template-based process (not described in this paper). Our experience has been that this is not an issue (as indicated in the evaluation results), and we argue that constructing a single plan block can be expected to be easier than constructing an entire program, i.e. that even where some plan blocks are not in the library, our process has the effect of reducing the problem to a smaller one. However, more broadly, this is a limitation of our work: we assume that the task at hand is reasonably well covered by the library of plans. Another limitation of this work is that the model of a plan network with data flowing between them is not expected to be applicable to all programming tasks. However, since our aim is to help novices to learn basic programming skills, we do not see the lack of universal applicability as a significant issue.

Having a programming process is, to some extent, a trade off in that the process is more structured (and hence more easily followed by novices), but also more complex than unguided programming. Our experience, and results, clearly show that the process is usable, and furthermore, that the benefits from having a structured process outweigh the costs of the additional complexity. We argue, as Kölling and Henriksen (2005) did, that without a programming process we could end up with two groups of students: those who fail programming, and those who pass by their own implicit process. We draw an analogy with swimming lessons. We would not let swimmers just jump into a river or the sea to learn swim. Instead, we prefer to teach them steps of swimming in a well-designed style at a swimming pool in the first place.

During the teaching, we recognised that some students were reluctant to follow the process they were taught (using goals and plans), and instead used BYOB blocks directly. It is not clear whether these students did not need the detailed process: they might be in the group who can find their own process implicitly (Kölling and Henriksen 2005). However, what is clear is that across the class, the new process did make a difference. It is possible that for some of the students the process assisted them to advance to a point where they no longer needed to follow the explicit process for simple programs. We also note that it may be easy to build up a program for a simple problem without explicit process, but it is hard to directly write a program for a complex problem.

There are a number of directions for future work. One direction concerns “bricoleurs” (Turkle and Papert 1990) who prefer to arrange and rearrange existing material transparently. To what extent does our process support or hinder this style of work? We argue that the steps of expanding and merging can actually expose and deal with the details of goals and plans, and that the feedback from each step should support a bricoleur style of negotiating and renegotiating. However, more work is needed to confirm this.

Another direction for further work concerns the number of plans. A limitation of our plan framework is that we only provided a limited number of plans. In other words, there are not many different plans to choose for the same goal. Therefore, one area of further investigation is to investigate students’ mental models in order to better understand how they select plans when many plans are available.

Finally, one weakness with the process is that the plan merging process, although well-defined, is somewhat complex. Therefore an area for future work is to investigate how to better support the plan merging process. However, we want to provide support that helps students to gain insight into the merging process, rather than just providing a “wizard” that does the merging of
plans. On the other hand, the expansion of plans (Section 3.3) can, and should, be automated.

6 References
Guzdial, M. & Soloway, E. (2002): Teaching the Nintendo generation to program, Communications of ACM, 45, 4, 17-21
Mayer, R. (1981): The psychology of how novices learn computer programming, Computing Surveys, 13, 1, 121-141, ACM